Konfigurations-Interoperabilität von Hardware-Software-Modellen in SystemC

Von der Carl-Friedrich-Gauß-Fakultät
Technische Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades
Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von

Dipl.-Ing. Christian Schröder
geboren am 20.01.1981 in Birkenfeld (Nahe)

Eingereicht am:	16. Mai 2011
Mündliche Prüfung am:	1. Juli 2011
Referent:	Prof. Dr. Ulrich Golze
Korreferent:	Prof. Dr.-Ing. Heinrich Theodor Vierhaus

(2011)

Kurzfassung

Im modernen Electronic System Level-Design wird zur Bewältigung der ständig steigenden Entwurfskomplexität auf hoher Abstraktion entwickelt. Unterstützung bieten dabei Systembeschreibungssprachen wie SystemC zusammen mit meist kommerziellen Entwicklungsumgebungen. Für eine hohe Entwurfseffizienz etwa bei der Architektur-Exploration sollten die Modelle untereinander über Herstellergrenzen und über Entwicklungsumgebungen hinweg problemlos austauschbar sein. Dafür sind Interoperabilitäts-Standards notwendig wie sie für SystemC bereits auf funktionaler Ebene existieren (TLM-2.0).

Über diese den realen Teil der Modelle betreffende Interoperabilität hinaus ist die Austauschbarkeit von Modellen zwischen Entwicklungsumgebungen bisher nicht standardisiert. Die vorliegende Arbeit definiert hierfür eine Meta-Interoperabilität und beschäftigt sich intensiv mit der dort einzugliedernden Interoperabilität der Konfiguration von Modellen. Es wird ein flexibler Konfigurationsmechanismus präsentiert, der existierende Mechanismen miteinander kompatibel macht und der in den von der Open SystemC Initiative (OSCI) erarbeiteten Konfigurations-Standard einfließt. Der Konfigurationsmechanismus basiert auf einer Modell-Middleware, die auch für weitere Ziele der Meta-Interoperabilität verwendet werden kann.

Abstract

In today's Electronic System Level Design, rapid platform development is done on a high abstraction level. Hence the system description language SystemC is getting used more and more often. For easy and fast architectural exploration, the designers must be able to integrate (high-level) models from different IP or model vendors as well as customized models into one platform. These models need to be interoperable with low effort. Therefore we need interoperability standards, like TLM-2.0 for SystemC which defines a communication standard for functional interoperablilty.

Another interoperability layer is the tool layer, which has not yet been addressed by a standard. This work defines meta interoperability and deales especially with the configuration interoperability as one part of meta interoperability. For model configuration a flexible interoperability framework is presented, which is able to connect different configuration mechanisms to each other. Additionally this work had been contributed to the upcoming configuration standard developed by Open SystemC Initiative (OSCI). Furthermore the configuration framework is based on a middleware which is applicable as a base for additional services for meta interoperability.

Danksagung

Die vorliegende Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter an der Abteilung Entwurf integrierter Schaltungen (E.I.S.) der TU Braunschweig. Hiermit möchte ich meinen Unterstützern Dank ausdrücken.

Ich danke meinem Betreuer und Mentor Prof. Ulrich Golze für seine sowohl fordernden als auch unterstützenden Worte und die Freiheiten, die er mir bei der Themenfindung gelassen hat, sowie für seine Geduld und Hilfe bei der Formulierung der Arbeit. Des Weiteren möchte ich meinem Koreferenten Prof. Heinrich Theodor Vierhaus für das entgegengebrachte Interesse an der Arbeit danken.

Großer Dank richtet sich an meine Kollegen und Freunde Wolfgang Klingauf, Robert Günzel und Hagen Gädke-Lütjens. Wolfgang hat mich in die Welt der akademischen Forschung eingeführt und hat mir in meiner Anfangszeit unschätzbar wichtige Impulse gegeben. Robert hat mir in unzähligen Stunden mit wertvollen Diskussionen, Review-Arbeit und Ablenkung („Lust auf Prokrastinieren?“) beigestanden. Das gegenseitige Beistehen in Phasen pessimistischer Anwandlungen war mir von außerordentlich hohem Wert.

Zum Dank verpflichtet bin ich auch Mark Burton, dem Gründer von GreenSocs. Er hat den Grundstein des in dieser Arbeit beschriebenen Projekts gelegt, das Interesse der Industrie daran geweckt und mir stets mit Feedback zur Seite gestanden. Weiterhin möchte ich den Studenten der Abteilung Entwurf integrierter Schaltungen und den GreenSocs-Mitarbeitern für ihre Beiträge an meiner Arbeit danken.

Tiefe Dankbarkeit empfinde ich gegenüber Melanie. Ihr Interesse an meiner Arbeit aus einer fachfremden Perspektive hat mir eine verständliche Präsentation des Themas ermöglicht. Ihr Beistand in allen Phasen der Arbeit ist von unschätzbarem Wert für mich. Ein besonderer Dank gilt meinen Eltern, die mich stets in allen Vorhaben unterstützt haben.

Konfigurations-Interoperabilität von Hardware-Software-Modellen in SystemC

Christian Schröder

Inhaltsverzeichnis

1. Einleitung

Inhalt

Arbeiten zum Hardware-Entwurf beginnen seit Jahrzehnten gern mit einem Zitat von Gordon Moore, der 1965 eine jährliche Verdopplung der Chip-Komplexität vorhersagte, der sicher aber selbst nicht geglaubt hat, dass sein *Moore'sches Gesetz* über 45 Jahre gültig bleiben würde[1] [ITRS10a]^i.

Diese explodierende Chip-Komplexität – heute ist sie bei unglaublichen 2 Milliarden Transistoren pro Chip angelangt – wurde auch zu einer ständig wachsenden Herausforderung an die menschlichen Entwickler, diesen elektronischen Überfluss mit möglichst sinnvollem Leben zu füllen. Die *International Technology Roadmap for Semiconductors* hat für diese neue Funktionsvielfalt den Begriff *More-than-Moore* geprägt [ITRS10b]^i. Er erweitert das Moore'sche digitale Wachstum um ebenfalls im Chip integrierte Interaktion mit der Außenwelt[2].

In der Folge fordern diese Trends auch immer neue und komplexere Entwurfsmethoden und Designtools. Zwei Ansätze bestehen in der *IP-Reuse* genannten Wiederverwendung von Hardware-Komponenten und im Einsatz *regulärer* Strukturen wie Speicher und Prozessorarrays, sodass weniger Hardware tatsächlich neu entwickelt werden muss. Das Ergebnis sind komplexe Hardware-Software-Systeme auf einem einzigen Chip, kurz Systems-on-Chip (SoC).

Zur Forschung der Abteilung Entwurf integrierter Schaltungen (E.I.S.) der TU Braunschweig gehören Methoden und Werkzeuge zum SoC-Entwurf, und in diesem Umfeld ist die vorliegende Arbeit angesiedelt.

Beispielsweise simuliert der virtuelle Prototyp VPOM-3430 [Syno11]^i die Hardware des System-on-Chip OMAP3430 [Texa11]^i von Texas Instruments in einer Entwicklungsumgebung von Synopsys. Mit diesem Prototyp kann Software bereits vor dem Abschluss des Hardware-Entwurfs entwickelt und so das traditionelle *Hardware-follows-Software* überwunden werden, das für die alles dominierende *Time-to-Market* so schädlich ist.

[1] Auch wenn besagte Verdopplung nur alle eineinhalb Jahre stattfand.

[2] Diese zusätzlichen Elemente können Sensoren, Aktoren, die Stromversorgung oder Weiteres sein, also beispielsweise analoge oder mechanische Komponenten. Ein Beispiel ist ein auf dem Chip integrierter Fotodetektor.

Im gerade genannten virtuellen Prototypen sind verschiedene abstrakte Modelle in der Systembeschreibungssprache SystemC zu einem System verbunden. Der Prototyp kann in der Entwicklungsumgebung um weitere kundenspezifische Hardwaremodelle erweitert werden. Leider hängt deren Entwurf derzeit völlig von der speziellen Entwicklungsumgebung ab. Möchte beispielsweise Motorola beim Entwurf des Droid-Mobiltelefons auf den Touch-Screen und die integrierte Digitalkamera anderer Firmen zurückgreifen, müssen diese ihre abstrakten Modelle für das Synopsys-Tool anpassen. Es liegt auf der Hand, dass alle Beteiligten Geld sparen und Fehler vermeiden könnten, wenn die Zulieferer ihre Modelle mit eigenen Tools entwickeln könnten und diese dann gleichwohl interoperabel mit allen Prototypen wären.

Die im Prototyp verwendete Systembeschreibungssprache SystemC bietet als ersten Schritt zur Interoperabilität den Kommunikationsstandard TLM-2.0 an. Wird das Touch-Screen-Modell TLM-2.0-konform entwickelt, kann es mit dem Prototyp kommunizieren. Für eine umfassende Integration in die Entwicklungsumgebung sind jedoch weitere Interoperabilitäts-Aspekte relevant, die aktuell nur mit proprietären Mechanismen möglich sind.

1.1. Zielsetzung

Für den Hardware-Software-Entwurf existieren diverse kommerzielle und freie Entwicklungsumgebungen mit der Systembeschreibungssprache SystemC. Typischerweise haben die Hersteller ihre Tools mit eigenen Erweiterungen ausgestattet, sodass SystemC-Modelle nicht zwischen verschiedenen Tools austauschbar sind. Eine solche Austauschbarkeit oder Interoperabilität ist aber wünschenswert, um die Wiederverwendbarkeit der Modelle zu erhöhen und den Entwicklungsaufwand zu verkleinern. Dieser Vorteil ist wie oben angedacht relevant, wenn die Modelle in einem anderen Projekt wiederverwendet, zwischen verschiedenen Herstellern ausgetauscht oder von externen Entwicklern eingesetzt werden sollen.

In der vorliegenden Arbeit beschäftige ich mich mit einem wesentlichen Aspekt von Interoperabilität: Der Nutzen eines Modells erhöht sich, wenn es *in konfigurierbarer Weise* für verschiedene Varianten eingesetzt werden kann. Beispielsweise könnten das Baudraten-Register eines UART-Hardwareblocks oder die Abstraktion der simulierten Kommunikation konfigurierbar sein. In der Regel bietet eine Entwicklungsumgebung jedoch nur einen eigenen proprietären Mechanismus an, was die Austauschbarkeit von Modellen zwischen Entwicklungsumgebungen behindert. Ziel meiner Arbeit ist die Verbesserung der Austauschbarkeit bezüglich der Konfiguration: Modelle aus verschiedenen Entwicklungsumgebungen sollen gemeinsam konfiguriert werden können.

Hauptziel und Ergebnisse

Diese Arbeit verfolgt als Hauptziel eine *Verbesserung der Austauschbarkeit von Modellen* zwischen verschiedenen kommerziellen und freien Entwicklungsumgebungen zur Erhöhung von Wiederverwendbarkeit und erreicht folgende Hauptergebnisse:

Ergebnis 1: Es wird ein *universeller Konfigurationsmechanismus* GreenConfig mit einer *Modell-Middleware* GreenControl erarbeitet.

Ergebnis 2: *Adapter für viele bereits existierende Konfigurationsmechanismen* integrieren Modelle in die universelle Entwicklungsumgebung von Ergebnis 1.

Ergebnis 3: Die Resultate und Erfahrungen meiner Arbeit fließen wesentlich ein in die Aktivitäten der Arbeitsgruppe OSCI CCI zur *Standardisierung*.

1.2. Relevanz und Industriekooperationen

Die Relevanz der Fragestellung und meiner Ergebnisse zeigten sich am akademischen und industriellen Interesse sowie den dabei entstandenen Projekten.

Viele der im Folgenden genannten Kooperationen wie auch das Zustandekommen des GreenControl- und GreenConfig-Projekts selbst wurden ermöglicht durch eine enge und erfolgreiche Kooperation mit der Firma GreenSocs. Der Gründer Mark Burton und die Abteilung E.I.S. haben sich zum Ziel gesetzt, die Industrie und die akademische Welt einander näher zu bringen. Es wurde eine anwendungsbezogene akademische Forschung erreicht, die entweder direkt von der Industrie unterstützt oder von ihr wahrgenommen wurde durch die auf der GreenSocs-Webseite zur Verfügung gestellte Informations-Plattform. Ein besonderes Augenmerk wurde auf die Open-Source-Eigenschaft der Projekte gelegt mit dem Ziel, der Community eine offen zugängliche Infrastruktur für die Modellierung von eingebetteten Systemen zur Verfügung zu stellen.

Das im Rahmen dieser Arbeit entstandene universelle Konfigurations-Framework GreenConfig wird mittlerweile von verschiedenen Firmen genutzt. Die Anforderungen und das Projekt sind wesentlich in Zusammenarbeit mit der Intel ESL-Modeling-Group aus Chandler, Arizona entstanden.

Industriell wird GreenConfig in unterschiedlichen Ausprägungen untersucht und genutzt von den Unternehmen Intel, Texas Instruments, Virtutech, CircuitSutra und Synopsys.

In Zusammenarbeit mit Texas Instruments (TI) ist eine Erweiterung für GreenConfig entstanden. Für den Einsatz bei TI wurde für die Scriptsprache Python eine GreenConfig-API (GreenScript) entwickelt. Synopsys hat eine Version der Innovator-Entwicklungsumgebung erstellt, die direkt GreenConfig-Parameter unterstützt. Die Firma Virtutech[3] hat ein Checkpoint-Verfahren für die Integration von SystemC in das Simics-Tool untersucht, das GreenConfig-Parameter verwendet. CircuitSutra[4] hat im Auftrag von GreenSocs und anderer Unternehmen diverse Modelle erstellt, die sich mit GreenConfig-Parametern konfigurieren lassen. Ein besonders wichtiges Beispiel dafür ist die virtuelle Plattform der Open-Core-Protocol-International-Partnership (OCP-IP) (vgl. Abschnitt 6.8.1). Diverse gut besuchte Präsentationen von Projekten (GreenControl, GreenConfig und GreenReg), die ich bei Firmen oder anlässlich von Konferenzen wie DATE oder DAC gegeben habe, belegen ebenfalls das

[3]Virtutech wurde im Februar 2010 von Wind River übernommen, das zu Intel gehört.
[4]Webseite CircuitSutra Technologies Pvt Ltd., India: `http://www.circuitsutra.com/`

industrielle Interesse an der hier präsentierten Arbeit. Der im folgenden Kapitel untersuchte Stand der Technik ist ein Indikator für die Relevanz dieser Arbeit. Er zeigt, dass noch kein universeller Ansatz für die Konfiguration existiert.

Auf akademischer Seite wurden erste Konzepte dieser Arbeit auf der internationalen „Conference on Hardware/Software Codesign and System Synthesis“ (CODES) 2009 in [SKG+09] veröffentlicht. Zudem gab es Kooperationen mit dem Institut CEPHIS der Universitat Autònoma de Barcelona. Das bereits erwähnte Checkpoint-Verfahren mit Virtutech wurde hier mit entwickelt. Meine vorläufigen GreenConfig-Performance-Messungen sind dort von Interesse gewesen und wurden auf der wissenschaftlichen Konferenz „Specification, Verification and Design Languages“ (FDL) 2009 und in einem Buchkapitel [MES+10] veröffentlicht. Auf internationales akademisches Interesse deutet eine Veröffentlichung auf dem PhD-Forum der Konferenz „Design, Automation & Test in Europe“ (DATE) 2011 [Schr11a] hin. In einem Projekt der Arbeitsgruppe von Prof. Berekovic (IDA, TU-Braunschweig) werden GreenConfig und weitere GreenSocs-Projekte für eine virtuelle Plattform verwendet[5].

Schließlich gibt es Bestrebungen der Open SystemC Initiative (OSCI), die Konfiguration von SystemC-Modellen zu standardisieren. Zu diesem Zweck wurde die Configuration, Control & Inspection Working Group (CCI WG) eingerichtet. In dieser Arbeitsgruppe arbeite ich mit, und die Ergebnisse und Erfahrungen dieser Arbeit fließen dort ein.

1.3. Konventionen in diesem Dokument

Dieses Dokument behandelt Themen, die wesentlich international geprägt sind und deren gebräuchliche Sprache Englisch ist. In dieser Arbeit werden vorwiegend deutsche Begriffe verwendet, sofern sie gleichbedeutend mit den etablierten englischen Begriffen existieren und gebräuchlich sind. In wenigen Fällen sind deutsche Begriffe entweder nicht vorhanden oder derart ungebräuchlich, dass dem sachkundigen Leser die Bedeutung verborgen bliebe. In diesen Fällen verwende ich englische Begriffe und erläutere deren Bedeutung, beispielsweise in einer Fußnote. Übersetzungen können auch in Klammern hinter dem Begriff stehen.

Der Schriftsatz dieser Arbeit richtet sich nach den folgenden Konventionen: Wichtige Begriffe werden bei ihrer ersten Verwendung **fett** gesetzt. Der Begriff wird an dieser Stelle erläutert und ist mit dieser Bedeutung im weiteren Verlauf der Arbeit reserviert. Besonders hervorzuhebende Zusammenhänge sind *kursiv* gesetzt. Programmcode-Auszüge werden sowohl in Listings als auch im Fließtext in `Typewriter` Schrift gesetzt.

Dieses Dokument lässt sich in Schwarz-Weiß-Druck ohne inhaltliche Einbußen betrachten, jedoch erhöht die farbige Darstellung stellenweise die Übersichtlichkeit. Einige Abbildungen zeigen Klassen- und Objektdiagramme. Diese sind möglichst nahe am UML-Standard 2.0 gehalten, weichen aber davon ab, wenn Modifikationen sinnvoll erschienen. Zudem sind die

[5] Webseite des Projekts für die European Space Agency (ESA):
`http://www.esa.int/TEC/Microelectronics/SEM151AMT7G_0.html`

Diagramme im Sinne der Übersichtlichkeit stellenweise vereinfacht und gekürzt. Vollständig sind sie in [Schr11b] abgebildet.

Zitate, die mit einem Suffix „i“ enden (zum Beispiel [ABCD09, EFGH08a][i]), stammen aus dem Internet. Ohne dieses Suffix stammen die Zitate aus der Literatur.

Dieses Dokument verweist auf externe Listings in der Art B.x (beispielsweise B.1). Es handelt sich meist um größere Code-Beispiele. Sie sind in Anhang B aufgeführt, wo jeweils detailliert auf eine externe Quelle verwiesen wird.

1.4. Aufbau dieses Dokuments

Das Einleitungskapitel 1 gibt einen groben thematischen Überblick über das Umfeld dieser Arbeit, ohne bereits alle verwendeten Begriffe ausführlich einzuführen oder zu definieren. Es bildet den Rahmen für die Arbeit und präzisiert die Zielsetzung und die Ergebnisse als grundlegende Herangehensweise. Zudem wird die industrielle und akademische Relevanz aufgezeigt.

Kapitel 2 legt die Grundlagen dieser Arbeit. Die Systembeschreibungssprache SystemC wird eingeführt, und Begriffe wie Konfiguration und Untersuchung werden definiert. Verschiedene Formen von Interoperabilität und Programmierschnittstellen werden vorgestellt. Der akademische und industrielle Stand der Technik wird erarbeitet.

Kapitel 3 analysiert den Stand der Technik und folgert Anforderungen an den in dieser Arbeit entwickelten universellen Konfigurationsmechanismus. Die Konfigurations-Interoperabilität für SystemC-Modelle wird eingeführt, und grundlegende Verfahren werden implementierungsunabhängig erläutert, welche zum Teil später realisiert werden.

Kapitel 4 führt die Konzepte und Details der Modell-Middleware GreenControl ein. Sie dient als flexible und erweiterbare Basis für den Konfigurations-Service und weitere Services wie den Analyse-Service in Anhang A.

Der Konfigurationsmechanismus GreenConfig wird als Middleware-Service in Kapitel 5 eingeführt. Hier wird die Realisierung der Konfigurations-Interoperabilität ausführlich erläutert. Verschiedene Integrationen ermöglichen die Adapter zu existierenden Mechanismen. Außerdem wird die Nutzung von GreenConfig für die Modell-Untersuchung erörtert.

In Kapitel 6 werden Anwendungsbeispiele für Realisierungen der zuvor eingeführten Konzepte diskutiert, die auch exemplarische Tests beinhalten. Zum einen werden Adapter für verschiedene Integrationen als Realisierungen der zuvor vorgestellten Konzepte im Detail erarbeitet. Zum anderen werden verschiedene Projekte vorgestellt, die den Konfigurationsmechanismus verwenden oder erweitern. Ein Abschnitt beschäftigt sich mit der Diskussion und Messung der Performance des vorgestellten Konfigurationsmechanismus.

Kapitel 7 fasst die Arbeit und ihre Ergebnisse zusammen und gibt einen Ausblick auf zukünftige Forschung.

Die Anhänge und Verzeichnisse können zum erweiterten Verständnis des Hauptteils herangezogen werden und schließen das Dokument ab.

2. Grundlagen

Inhalt

Eingebettete Systeme werden als immer komplexere Hardware-Software-Codesigns entwickelt. Da die Design-Effizienz nicht in gleichem Maße ansteigt, entsteht ein *Design-Gap*, das es zu schließen gilt [Golz09]. Gleichzeitig gibt es eine Vielzahl Anforderungen an das Design, beispielsweise an die Performance, den Energieverbrauch, die Time-to-Market[1] und die Herstellungskosten. Eine primäre Herausforderung und Aufgabe des Systemdesigners ist es, die Architektur des Systems optimal auszulegen, also den verschiedenen Anforderungen gerecht zu werden – zum Beispiel die optimale Balance zwischen Performance und Energieverbrauch zu finden. Die herkömmlichen Methoden für das Systemdesign abstrahieren kaum und sind mit der steigenden Komplexität der Systeme immer schlechter anzuwenden, da sie zu wenig globalen Einblick in das Gesamtsystem erlauben bzw. die zur Verfügung stehenden Methoden zu detailliert sind, um sinnvoll eine Systemanalyse und -optimierung durchführen zu können [BaMA07].

Demnach ist es zum Schließen des Design-Gap hilfreich, die Entwicklung von Systemen auf einer höheren Abstraktionsebene (Electronic System Level (ESL)) durchzuführen, was sich in der Industrie nach und nach mit verschiedenen Entwurfsabläufen durchsetzt. Das ESL-Design erlaubt eine abstrakte Modellierung, wobei der Grad der Abstraktion variieren kann. Das zu entwickelnde System wird hier als – im Vergleich zu einem Modell auf Register-Transfer-Ebene (RTL) – *abstraktes Modell* dargestellt. Ein wesentlicher Schritt der Abstraktion ist die Aufteilung des Systems in Kommunikation und Verhalten. Die Kommunikation wird dann mit Hilfe von Transaktionen modelliert.

[1] Produkteinführungszeit

Transaction-Level-Modellierung (TLM) Mit der Transaction-Level-Modellierung (TLM) wird das Designproblem in beherrschbare Teilprobleme geteilt: Kommunikation und Funktion werden unabhängig voneinander modelliert („orthogonal“). Zudem können Kommunikation und Funktion auf jeweils unterschiedlichen Abstraktionsdomänen modelliert werden [Golz09]. Ein Einordnungsschema für die jeweils unterschiedlichen Abstraktionsdomänen für Kommunikation und Funktion wird in [CaGa03] vorgeschlagen.

Systembeschreibungssprachen Für das ESL-Design sind verschiedene Systembeschreibungssprachen entwickelt worden. Prominente Vertreter sind SpecC [Gers01], SystemVerilog [IEEE09] und SystemC [IEEE06]. Von diesen Sprachen hat sich SystemC durchgesetzt [BaMA07], sie ist die für TLM vorherrschende Sprache.

2.1. SystemC

SystemC [IEEE06] ist eine von der Open SystemC Initiative (OSCI) entwickelte Sprache, die seit der Version 2.0 für die Systembeschreibung geeignet ist. Zunächst ist sie in der Version 1.0 als RTL-Sprache entwickelt worden, die aktuelle Version ist 2.2. Interfaces und Channels sowie die Erweiterung um TLM mit dem Standard TLM-2.0 [OSCI09] unterstützen die orthogonale Modellierung von Kommunikation und Funktion.

SystemC ist eine Klassenbibliothek für C++ [ISO03] und erbt dessen Objektorientierung, was ein Alleinstellungsfaktor gegenüber Hardwarebeschreibungssprachen ist. SystemC erweitert C++ um Konstrukte zur Modellierung von Zeit, Parallelität und hardwarenahen Datenstrukturen. Abbildung 2.1 zeigt eine Übersicht über die SystemC-Architektur. Bemerkenswert ist besonders im Hinblick auf Konfigurierbarkeit, dass die Datentypen vom Sprachkern getrennt sind, es können also auch C++- oder eigene Datentypen verwendet werden.

Neben dem SystemC-Standard bietet die Non-Profit-Organisation OSCI auch einen frei verfügbaren Open-Source-Simulator an. Wenn nicht anders erwähnt, wird in dieser Arbeit dieser Simulator (der SystemC-Version 2.2) verwendet. Unberührt davon ist der hier entwickelte Basismechanismus vom Simulator unabhängig, funktioniert also auch mit standardkonformen kommerziellen Simulatoren.

2.1.1. Phasen der SystemC-Ausführung

Während des Programmablaufs einer SystemC-Simulation werden verschiedene Phasen durchlaufen. Die Phasen und Zeitpunkte sind für das Anwenden der Konfiguration von Bedeutung. Hier folgt eine Übersicht über die verschiedenen Phasen und eine Einführung von notwendigen Begriffen (vgl. SystemC-Standard [IEEE06]), eine Übersicht gibt Abbildung 2.2:

Eine **SystemC-Anwendung** ist SystemC-Code, der die standardisierte SystemC-Bibliothek verwendet. Für eine Simulation unter Anwendungskontrolle (vgl. Application Control [IEEE06], Abschnitt 4.3) ist die Funktion `sc_main` der Einstiegspunkt in eine SystemC-Anwendung. Sobald diese Funktion aufgerufen ist, befindet sich die Simulation laut Standard in

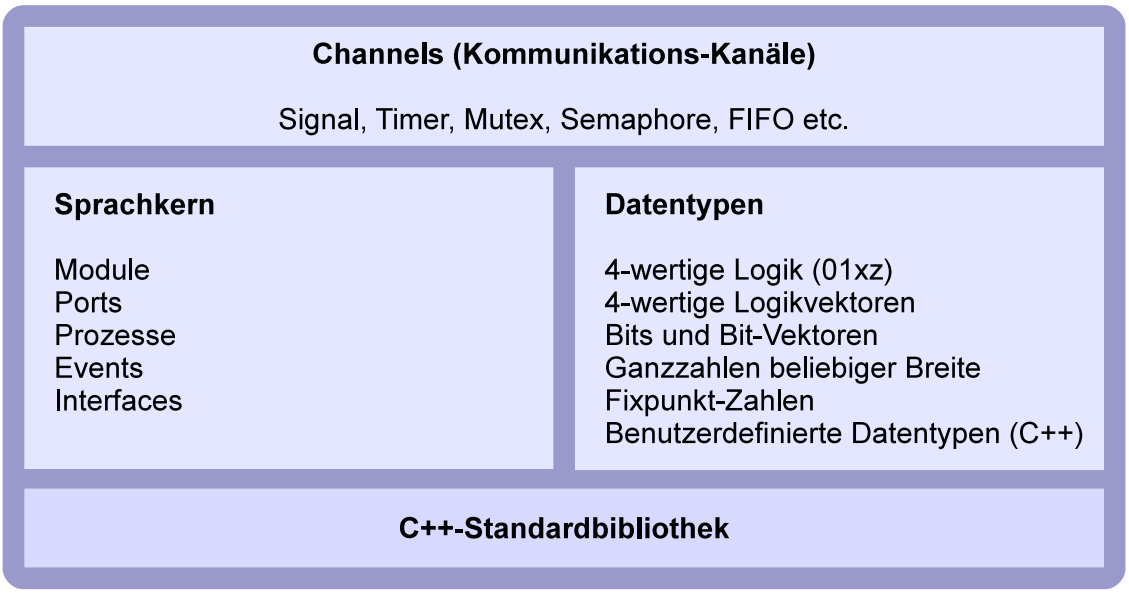


Abbildung 2.1.: SystemC-Architektur: Die Datentypen sind vom Sprachkern getrennt. (nach [GLGS02])

der **Elaborationsphase**. Diese Phase dient den Vorbereitungen und dem Aufbau des zu simulierenden Systems. Durch den ersten Aufruf der Funktion `sc_start` wird die Elaborationsphase unmittelbar beendet. Das wird durch die Callbacks `before_end_of_elaboration` und `end_of_elaboration` angezeigt. Anschließend beginnt die **Simulationsphase (SimPh)**, angezeigt durch den Aufruf des Callbacks `start_of_simulation`. Die Simulationsphase kann aus wiederholten Aufrufen der Funktion `sc_start` bestehen. In der Simulationsphase schreitet die simulierte Zeit for. Das Ende der Simulationsphase wird durch den Callback `end_of_simulation` angezeigt, wenn die SystemC-Anwendung die Funktion `sc_stop` aufruft – was nicht zwingend gefordert ist. Die Simulationsphase ist zu Ende, wenn die Funktion `sc_main` (zum letzten Mal) zurückkehrt.

Die Kombination aller Phasen, also Elaborations- und Simulationsphase, in denen die SystemC-Anwendung ausgeführt wird, plus die umgebende SystemC-Implementierung werden **Programmphase** genannt. Im Falle des OSCI-Simulators ist die Ausführung der Programmphase die tatsächliche Laufzeit der ausführbaren kompilierten Programmdatei. Bei anderen Simulatoren, die in kommerziellen Entwicklungsumgebungen zum Einsatz kommen, können zur Programmphase weitere herstellerspezifische Aktionen gehören. Für die Konfiguration von Modellen ist das von Bedeutung, da es auch Elementen dieser Entwicklungsumgebungen möglich sein muss, auf die Eigenschaften des Modells lesend oder schreibend zugreifen zu können.

2.2. Die Begriffe „real“ und „meta“

Einige weitere Begriffe sind für die in dieser Arbeit betrachtete Konfiguration von Bedeutung:

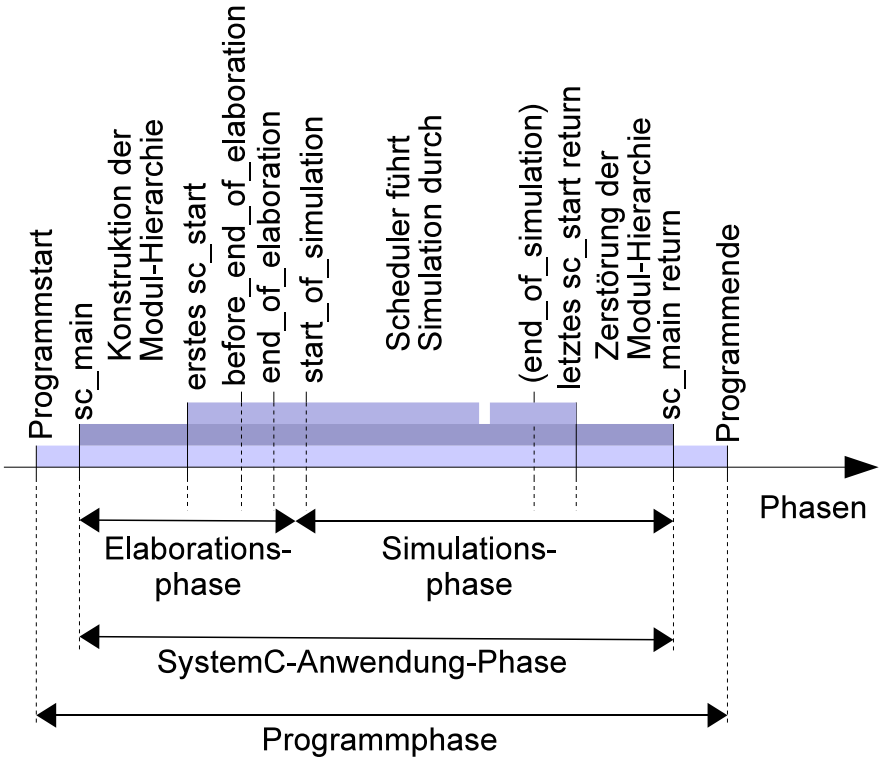


Abbildung 2.2.: SystemC Phasen- und Zeitbegriffe

Ein **Real-Modell** ist ein in SystemC abgebildetes Hardware- oder Software-Modell. Es handelt sich ausschließlich um den Teil des Modells, der zu einem späteren Zeitpunkt realisiert werden soll. Das sind zum Beispiel Statemachines, Register, Kommunikationsstrukturen, Prozessoren sowie Software.

Meta-Informationen und **Meta-Funktionalität** sind zusätzliche Elemente, die ein Real-Modell um Informationen und Funktionen erweitern und die für den Entwickler oder Nutzer des Modells nützlich sind zum Dokumentieren, Debuggen, Verwalten, Konfigurieren, Analysieren etc. Sobald ein Modell durch Hinzufügen von Meta-Informationen oder -Funktionalität kein reines Real-Modell mehr ist, ist es ein **Meta-Modell** (vgl. Abbildung 2.3). Folglich beinhaltet ein Meta-Modell im Normalfall eine Klasse von Real-Modellen. Die Grenze zwischen den Elementen des Real-Modells und denen des Meta-Modells sind unter Umständen schwierig festzulegen, beispielsweise wenn Klassen oder Objekte verwendet werden, die Real-Modell-Elemente repräsentieren und gleichzeitig Meta-Zugriff, z.B. das Auslesen zu Debug-Zwecken, erlauben. Der Unterschied lässt sich nicht syntaktisch feststellen oder definieren. Die Unterscheidung muss im Einzelfall angegeben werden, ist in den meisten Fällen aber offensichtlich.

Die oberste Ebene der Simulation kann mit einer Testbench realisiert sein, die eine oder mehrere Modell-Instanzen erstellt und möglicherweise miteinander verbindet. Eine **Real-Testbench** beschränkt sich auf die klassische Stimulation mit Inputs für das Modell oder einer In-the-Loop-Verschaltung verschiedener Modelle mit ausschließlich die Realität widerspiegelnden In- und Outputs. Eine **Meta-Testbench** verbindet die (Real-/Meta-)Modelle

miteinander und führt administrative Arbeiten wie die Konfiguration verschiedener Szenarien, Untersuchung und Analyse von Ergebnissen usw. durch. Die Verifizierung der Outputs in einem In-the-Loop-Aufbau gehört somit auch zur Meta-Testbench. In einer nur vom Kernel kontrollierten Simulation (d.h. es gibt keine `sc_main`-Funktion) kann ein Simulator oder eine integrierte Entwicklungsumgebung (IDE) fordern, dass die Testbench ein Top-Level-Modul ist.

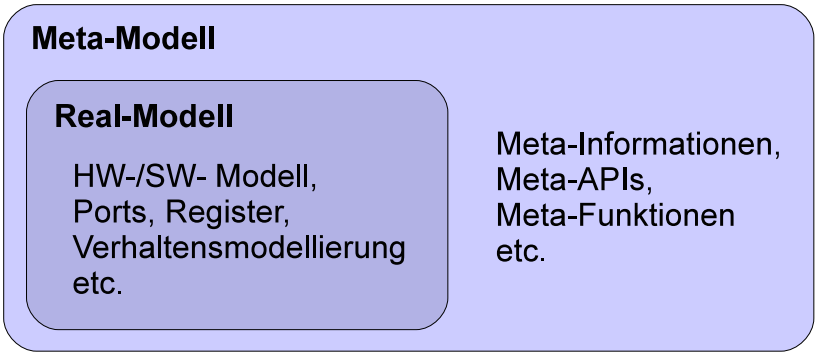


Abbildung 2.3.: Modell-Begriffe

2.3. Konfiguration und Untersuchung

Modell-Konfiguration ist die Kombination aus den zwei folgenden Spezialfällen und bedeutet im hier betrachteten Fall, Eigenschaften eines Modells zu beeinflussen.

Klassische Konfiguration bedeutet, Eigenschaften eines Modells während der Programmphase, aber noch *vor der Simulationsphase* (vgl. Abschnitt 2.1.1) zu beeinflussen. Es ist denkbar, dass beispielsweise eine Konfigurationsdatei vom Benutzer noch vor dem Start der Programmphase erstellt wird, das Einlesen dieser Daten findet aber durch das Programm während der Programmphase statt. Dadurch zählt diese Art der Konfiguration auch zur klassischen Konfiguration.

Ebenfalls unter den Begriff der Modell-Konfiguration wird der Fall eingeordnet, dass die Beeinflussung von Konfigurationseigenschaften *während der Simulationsphase oder danach* vorgenommen wird. Diese Unterart der Modell-Konfiguration wird **Modell-Steuerung** genannt. Nach der Simulationsphase kann die Modell-Steuerung beispielsweise Einfluss auf Werte zur Debugausgabe von Ergebnissen nehmen.

Diese Definition impliziert eine Beschränkung der Konfiguration auf die Beeinflussung von Modellen nach dem Kompilieren des Quellcodes. Manipulationen am Quellcode wie beispielsweise Template-(Schablonen-)Parameter[2] und Präprozessor-Makros (`#define`) werden folglich nicht als Konfiguration gewertet. Diese Einschränkung ist notwendig und sinnvoll, da andernfalls konsequenterweise jede manuelle Quellcode-Manipulation (z.B. Austauschen

[2]Template-Parameter sind Schablonen für C++-Klassen, die vom Compiler ausgewertet werden.

eines +-Operators durch einen ∗-Operator) als Konfiguration angesehen werden müsste[3]. Tatsächlich wird in diesen Fällen aber ein neues (potentiell konfigurierbares) Modell geschaffen.

Das Auslesen von Eigenschaften zu beliebigen Zeitpunkten heißt **Modell-Untersuchung**. Bei den ausgelesenen Eigenschaften kann es sich hierbei um Konfigurations-, Kontroll- oder andere nur für Untersuchungszwecke existierende Eigenschaften handeln. Die Untersuchung kann auch nach der Programmphase fortgesetzt werden, indem beispielsweise Dateien ausgewertet werden, die während der Simulationsphase geschrieben wurden.

Das Setzen und Auslesen von Eigenschaften kann im Allgemeinen sowohl innerhalb als auch außerhalb von Meta-Modellen erfolgen. Meta-Modelle können sich gegenseitig konfigurieren und untersuchen, sowie von der umgebenden Testbench oder sogar von der IDE aus.

Abbildung 2.4 verdeutlicht die Begriffe.

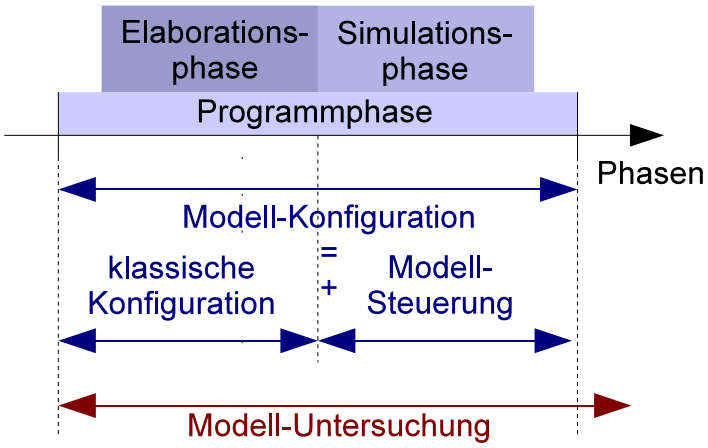


Abbildung 2.4.: Konfigurationsbegriff

Der in dieser Arbeit vorgestellte Mechanismus erlaubt sowohl Modell-Konfiguration als auch Modell-Untersuchung. Da speziell das Untersuchen von Modelleigenschaften zur Simulationslaufzeit sehr häufig stattfindet, sollte der Mechanismus effizient sein.

2.4. Interoperabilität

Für Intellectual Property (IP) auf der RTL ist es Realität, IP in Form von vorgefertigten Hardware-Elementen (Cores) verschiedener Hersteller in einem Projekt zu kombinieren. Einerseits wird diese Modul-Interoperabilität durch streng definierte Kommunikations-Interfaces auf RTL-Pin-Level ermöglicht, vorausgesetzt die Cores nutzen ein gemeinsames Interface, z.B. einen On-Chip-Bus. Wenn dieses nicht der Fall ist und der Core tatsächlich auf dem Chip verwendet werden soll, ist es andererseits vertretbar, einen Umsetzer (Wrapper) zu entwerfen.

[3]Das Setzen der Busbreite in TLM-2.0 mit einem Template-Parameter ist folglich leider keine Konfiguration.

Auch beim Systementwurf und der Simulation mit SystemC ist diese Art von Interoperabilität wünschenswert. Allerdings soll während des Systementwurfs schnell und unkompliziert erprobt werden, welchen Aufbau der Chip später haben soll (Architektur-Exploration). Deswegen müssen Modelle verschiedener Hersteller in einer Simulation zusammen verwendet werden können, ohne dass zeitaufwändig Umsetzer entworfen werden müssen, die die Modelle zusammen arbeiten lassen. Es ist hier nicht vertretbar, für viele ausprobierte Modelle zunächst einen Wrapper für die Verbindung zu programmieren. Da SystemC aber nicht auf bestimmte Abstraktionsstufen der Abstraktionsdomänen festgelegt ist (vgl. [BaMA07, S. 224]), ist es ungleich schwieriger, Regeln für die Interoperabilität von Modellen zu definieren.

Für den Bereich der Kommunikation werden von der OSCI mit dem TLM-Standard [OSCI09] Anstrengungen unternommen, Regeln für die Kommunikation zu standardisieren. Ein Standardisierungsvorschlag für taktgenaue TLM-Simulation beschreibt Günzel in [Günz11]. Diese Kommunikations-Standardisierung ist ein wichtiger Schritt, Real-Modelle in einer Simulation interoperabel zu machen. Zusätzlich sind weitere Aspekte denkbar, die ein Zusammenarbeiten von Modellen beeinflussen.

Man kann zwei verschiedene grundsätzliche Interoperabilitäts-Ebenen unterscheiden:

- Die **Real-Interoperabilität** beinhaltet sämtliche Aspekte, die notwendig sind für die Zusammenarbeit von mindestens zwei Real-Modellen in einem übergeordneten Real-Modell. Zwischen TLM-Modellen wird diese Interoperabilität durch die klassische Interoperabilität in der Kommunikation sichergestellt.

- Die **Meta-Interoperabilität** beschreibt die Beziehung zwischen (mindestens) einem Tool und (mindestens) einem Modell und beinhaltet die Aspekte, die die Simulation selbst betreffen, also die Aspekte, die ein Meta-Modell im Gegensatz zu einem Real-Modell ausmachen und sich nach der Produktion nicht auf dem Chip wiederfinden und für ein Zusammenarbeiten zwischen dem Tool und dem Modell relevant sind, beispielsweise die Einbindung des Modells in die Entwicklungsumgebung.

Das Hauptziel dieser Arbeit ist es, diese Meta-Interoperabilität zu verbessern, um die Zusammenarbeit von Modellen mit verschiedenen IDEs und damit die Austauschbarkeit der Modelle zu erhöhen, um eine schnelle Architektur-Exploration zu ermöglichen. Der Fokus dieser Arbeit liegt besonders auf der Modell-Konfiguration.

2.5. Programmierschnittstellen

Ein **Application Programming Interface**, oder eine **Programmierschnittstelle (API)**, ist eine Sammlung von Funktionen und Definitionen von Datenstrukturen, die den Zugriff auf eine Funktionsbibliothek ermöglichen [PaHe08, S. A-4].

Im Bereich der Modell-Konfiguration gibt es zwei sinnvoll voneinander abzugrenzende APIs:

Die **Modell-API** ist die innerhalb des konfigurierbaren Modells verwendete Programmierschnittstelle, unabhängig davon, ob es ein Real- oder Meta-Modell ist. Sie erlaubt es, ein Modell konfigurierbar zu machen und bietet potentiell eine komfortable Verwendungsmöglichkeit konfigurierbarer Elemente innerhalb des Modells an. Da die Modelle in SystemC modelliert sind, ist die Modell-API hier stets eine C++-API.

Dagegen bezeichnet man alle weiteren Konfigurations-Schnittstellen als **Tool-API**. Das können einerseits weitere C++-APIs sein, die den Tools den Zugriff auf die konfigurierbaren Elemente von Modellen erlauben, andererseits aber auch Schnittstellen zu anderen Programmiersprachen oder grafische Schnittstellen zum Benutzer, die das Betrachten oder Setzen der konfigurierbaren Elemente ermöglichen. Es kann sinnvolle Fälle geben, in denen sogar ein Modell selbst eine Tool-API verwendet, um beispielsweise Eigenschaften von Untermodulen zu konfigurieren.

Im Rahmen der in Kapitel 4 eingeführten Modell-Middleware werden GreenControl-User-APIs (kurz: User-API) als Programmierschnittstelle zu den Middleware-Services eingeführt. Im Fall der Konfiguration ist eine User-API entweder eine Modell-API oder eine Tool-API.

2.6. Stand der Technik

Der in dieser Arbeit entwickelte Konfigurations- und Untersuchungsmechanismus soll Adapter zu vielen anderen Mechanismen und Tools anbieten und Merkmale dieser Mechanismen unterstützen. **Merkmale** sind dem Benutzer zur Verfügung gestellte, besonders hervorzuhebende Programmeigenschaften. Um zu klären, welche das namentlich sind, wird zunächst der Stand der Technik erarbeitet. Er ergibt sich aus akademischen und industriellen Ansätzen. Im folgenden Kapitel werden aus einer Analyse dieser Related Work Anforderungen (Requirements) für den eigenen Mechanismus abgeleitet.

Im Abschnitt 2.6.1 werden akademische Arbeiten aufgezeigt, die sich mit einem ähnlichen Thema oder einem Teilaspekt dieser Arbeit beschäftigen. Die Anzahl der für das Thema bedeutungsvollen Bibliotheken und Tools ist begrenzt, was sich durch den speziellen Fokus und die Aktualität und Industrierelevanz des Themas erklären lässt.

Aus diesem Grund werden für den Stand der Technik weitere vor allem für die Industrie bedeutungsvolle Mechanismen untersucht. Im Abschnitt 2.6.2 werden Ansätze vorgestellt, die nicht rein akademisch sind, aber durch die Mitwirkung mehrerer Firmen und Universitäten von übergeordneter Bedeutung sind.

Relevant für diese Arbeit sind insbesondere die verschiedenen bereits existierenden Mechanismen zur Modell-Konfiguration von SystemC-Modellen, die meist von kommerziellen Herstellern in ihren Tools vertrieben werden. In Reviews werden einige von ihnen exemplarisch diskutiert. Deren Relevanz definiert sich dadurch, dass diese Mechanismen einerseits die Basis für die Analyse des Themas bilden (vgl. Abschnitt 3.1) und andererseits die Mindestanforderungen an den in dieser Arbeit entworfenen Mechanismus stellen (vgl. Abschnitt 3.2). Die untersuchten Mechanismen finden sich in der CoWare SystemC-Modeling-Library (SCML),

dem ARM Cycle Accurate Simulation Interface (CASI), den Synopsys CCSS-Parametern, der Open-Verification-Methodology (OVM), Texas Instruments (TI) SystemPython und dem Intel Device Register Framework (DRF).

2.6.1. Akademische Ansätze

In der Mehrzahl der Fälle ist das Ziel der im Folgenden diskutierten akademischen Ansätze die Untersuchung eines SystemC-Modells auf hohem Systemlevel. Zu diesem Zweck wird das Modell mit der klassischen Konfiguration konfiguriert. Einige Arbeiten setzen ihren Fokus auch auf die gesamte Modell-Konfiguration während der Simulationsphase zu Debug-Zwecken.

In [ChMa05] werden einige grundlegende Möglichkeiten aufgezählt, mit denen auf Basis von C bzw. C++ SystemC-Modelle auf manuelle Art und Weise verändert werden können, was die Autoren „Konfiguration in SystemC“ nennen. Das sind Codegenerierung, Makros, Linken von verschiedenen Objektdateien, Templates, Polymorphismus und einfache Kontrollanweisungen im Code. Fünf von sechs Verfahren manipulieren den Quellcode vor oder während des Kompilierens oder des Linkens oder erzeugen ein neues Modell, sodass sie nicht der Definition von Modell-Konfiguration entsprechen. Lediglich die Kontrollanweisungen im Code verändern das Verhalten während der Programmphase, sind aber trivial und deswegen nicht von Interesse.

In [ARA+07] wird ein Mechanismus vorgestellt, der eine C++-Reflection-Bibliothek für die dynamische Analyse und Steuerung verwendet. Die Autoren fokussieren ihre Arbeit auf das Debuggen von SystemC-Plattformen, die nicht im Quelltext vorliegen müssen und die vom Entwickler nicht mit speziellen Debug-Fähigkeiten ausgestattet werden müssen. Diese nicht SystemC-spezifische Lösung kann aus einer Header-Datei und der Objekt-Datei ohne neues Kompilieren im Objekt verwendete Parameter zur Analyse lesen und zum Kontrollieren schreiben.

Für jedes SystemC-Modul, das untersucht oder kontrolliert werden soll, wird eine sogenannte WhiteBox in die Simulation eingefügt. Diese Box erhält beim Konstruieren einen Pointer auf das zu untersuchende Modul und wird mit der gleichen Aktivierungsliste (Sensitivität) versehen. Dadurch werden die Untersuchungszeitpunkte auf diejenigen beschränkt, an denen der SystemC-Scheduler das untersuchte Modul aktiviert oder auf beobachteten Werten basierende Breakpoints erreicht werden. TLM-2-Kommunikation ohne Events scheidet folglich aus, was inakzeptabel ist. Leider ist die verwendete Entwicklungsumgebung nicht mehr verfügbar.

Im Gegensatz zum in dieser Arbeit angestrebten Konfigurationsmechanismus braucht sich der Modellentwickler keine Gedanken über die Konfigurierbarkeit oder Analysierbarkeit seines Modells zu machen. Dadurch bleibt es allerdings auch dem Benutzer des Debug-Mechanismus überlassen, sinnvolle Parameter zu finden, möglicherweise sogar ohne den Quellcode zu kennen. Da der Fokus das Debuggen ist, ist zudem die Steuerung des Modells nur dyna-

misch durch den Benutzer zur Simulationsphase vorgesehen, sodass keine Initialkonfiguration vorgegeben werden kann.

Wie [ARA+07] verwendet auch die Simulationsplattform ReSP [BBF+08] einen Reflection-orientierten Ansatz. Die Multi-Prozessor-Entwicklungsumgebung beinhaltet die Unterstützung von Python-Skripten, z.B. für dynamisches Verbinden des Modells ohne C++-Compiler-Durchlauf und das dynamische Untersuchen und Steuern des SystemC-Modells während der Simulationsphase. Zudem bietet der ReSP-Kern eine Simulationskontrolle an, vergleichbar mit der in Abschnitt 6.7 vorgestellten. Das für die Design-Space-Exploration genutzte SystemC-Modell kann auch hier nicht für die Analyse oder Konfiguration ausgestattet werden, was in dieser Arbeit aber gefordert ist.

Weitere akademische Arbeiten sind gSysC [AlEH06] und [RGDR07]. Sie präsentieren Entwicklungsumgebungen mit grafischer Aufbereitung von Analyseergebnissen und decken auch Teile der Modell-Konfiguration bzw. Modell-Untersuchung ab.

Die Integrated-Development-Environment [RGDR07] legt besonderen Wert auf die Visualisierung. Ein integrierter Debugger erlaubt sowohl das klassische Debuggen auf der C++-Ebene (mit GDB), als auch das Debuggen auf Systemebene, also beispielsweise das Steuern der Simulation, das Setzen von High-Level-SystemC-Breakpoints und das Untersuchen von SystemC-Elementen wie Signalen, Events und Ports. Die Visualisierung kann Änderungen sogar während der Simulationsphase anzeigen. Die IDE verwendet den unmodifizierten OSCI-Kernel und unmodifizierte Bibliotheken, indem eine dynamisch gelinkte Bibliothek die originalen (z.B. Kernel-)Funktionen überschreibt.

Auch gSysC [AlEH06] ist eine grafische IDE mit Simulationskontrolle, grafischer Modell-Untersuchung und konditionalen Breakpoints. Der Ansatz von gSysC behält den SystemC-Kernel bei und stellt die Verbindung zur Simulation über Makros her, die der Entwickler in den Modell-Quellcode für diejenigen Elemente einbaut, die untersucht und angezeigt werden sollen. Die Simulationskontrolle wird durch Überschreiben von SystemC-Funktionen ermöglicht. Modell-Konfiguration ist nicht vorgesehen.

In [PMBS06] wird eine auf CORBA[4] basierende Service-orientierte Architektur für SystemC vorgestellt. Als ein Service wird auch der Reflection-Service präsentiert, der Modell-Untersuchung unterstützt. Die Autoren geben keine Informationen, ob auch nicht-SystemC-Datentypen unterstützt werden. Ein Service für Modell-Konfiguration wird nicht vorgestellt. Zudem wird ein Performance-Einbruch von etwa 50% gemessen, der zwar nach Aussage der Autoren akzeptabel ist, für den in dieser Arbeit entwickelten Mechanismus für Modell-Untersuchung während der Simulationsphase aber nicht akzeptabel ist[5].

Zusammenfassend präsentieren die meisten Autoren entweder ein Konzept zum Debuggen, d.h. Modell-Untersuchung, teilweise mit Modell-Konfiguration inklusive Modell-Steuerung, oder rudimentäre klassische Konfigurationsmechanismen vor der Simulationsphase. Ersteres

[4] Common Object Request Broker Architecture (CORBA) ist ein mächtiger Software-Middleware-Standard, siehe `http://www.omg.org/technology/documents/corba_spec_catalog.htm`.

[5] Diese Aussage betrifft die Modell-Untersuchung, nicht die klassische Konfiguration, die nicht derart Performance-kritisch ist, vgl. Abschnitt 6.9.

erlaubt es dem Entwickler nicht, bestimmte Elemente des Modells als besonders empfehlenswert für die Modell-Konfiguration hervorzuheben, letzteres erlaubt keine Modell-Steuerung. Da die Mechanismen primär auf das Debuggen zielen, ist es einem Modell nicht möglich, während der Simulationsphase auf die Modell-Steuerung zu reagieren.

2.6.2. Ansätze übergeordneter Bedeutung

Im Folgenden werden zwei firmenübergreifende Ansätze vorgestellt. Sie sind nicht rein akademisch, haben aber durch das Mitwirken von verschiedenen Firmen und Universitäten eine den später vorgestellten proprietären Ansätzen übergeordnete Bedeutung.

OSCI Configuration, Control & Inspection (CCI) Working Group

Da ich in der Arbeitsgruppe mitarbeite, bildet die OSCI Configuration, Control & Inspection Working Group (CCI WG) (kurz: CCI)[6] einen Schwerpunkt dieser Arbeit. Die Diskussionen in der Arbeitsgruppe haben diese Arbeit beeinflusst, beispielsweise in den Anforderungen. Zudem sind einige Erkenntnisse, die ich im Laufe dieser Arbeit gewonnen habe, in die Arbeitsgruppe und den dort entwickelten Prototypen getragen worden. Aufgrund dieser Sonderstellung nimmt diese Arbeit im Folgenden mehrfach Bezug auf die CCI.

Die CCI ist eine Arbeitsgruppe der OSCI, die einen neuen Standard entwickelt, der die Interoperabilität zwischen Modellen und Tools verbessern soll. Der Standard soll beschreiben, wie Modelle instrumentiert werden können, damit sie in unterschiedlichen Tools konfiguriert (klassische Konfiguration), gesteuert (Modell-Steuerung) und analysiert (Modell-Untersuchung) verwendet werden können. Damit verfolgt er eines der Ziele dieser Arbeit, weswegen ich in der Gruppe mitarbeite. Die CCI bewältigt ihre Arbeit inkrementell und hat mit dem Konfigurationsaspekt begonnen. Die Arbeitsgruppe profitiert von der aktiven Mitarbeit vieler einflussreicher Firmen, beispielsweise ARM, Cadence, Carbon Design Systems, CoWare, Intel, Mentor Graphics, NXP, STMicroelectronics, Synopsys, Texas Instruments und Virtutech.

Meine Mitarbeit in der Arbeitsgruppe erfolgte im Rahmen meiner Forschungstätigkeit an der Abteilung E.I.S. der TU Braunschweig. Viele interessante Diskussionen über Use-Cases, Anforderungen und die zu standardisierenden Schnittstellen haben zu einem Prototyp geführt. Bis zum heutigen Datum wurde er federführend von mir entwickelt und implementiert die gemeinsam in der Arbeitsgruppe entstandene API für den Standard. Aktuell verwendet dieser Prototyp intern den in dieser Arbeit entwickelten universellen Mechanismus, was neue Anforderungen an diesen gestellt hat und nun zeigt, dass er den Anforderungen des neuen Standards gewachsen ist.

Ein nach außen sichtbares Ergebnis der Arbeitsgruppe ist das Anforderungsdokument [OSCI09, OSCI10]^{i}, das im Februar 2010 für das öffentliche Review publiziert wurde. Dieses Dokument wird auch bei der Aufstellung der Anforderungen im Abschnitt 3.2 berücksichtigt.

[6] OSCI CCI Webseite: `http://www.systemc.org/apps/group_public/workgroup.php?wg_abbrev=cciwg`

SPIRIT IP-XACT

Das SPIRIT Consortium hat die IP-XACT-Spezifikation [SPIR08] entwickelt. Diese Spezifikation ist fokussiert auf IP-Wiederverwendbarkeit und spezifiziert dafür ein Standard-Austauschformat für IP-Interface-Beschreibungen. Das beinhaltet auch die statische Konfiguration des Modell-Quellcode. Konkret werden zu diesem Zweck Schnittstellen für IP-Generatoren beschrieben, die Quellcode abhängig von der Konfiguration generieren können. Es handelt sich nach der Definition dieser Arbeit nicht um Modell-Konfiguration, sondern um das Generieren verschiedener Modelle, trotzdem erscheint es sinnvoll, eine mit IP-XACT beschriebene Konfiguration in den hier entwickelten Mechanismus übernehmen zu können.

2.6.3. CoWare SystemC-Modeling-Library (SCML)

CoWare bietet die Toolsammlung *Platform-Architect* an und bewirbt sie mit „SystemC Platform Capture and Architecture Analysis for Electronic System Virtualization“ (vgl. [CoWa10a]^i)[7].

Der Platform-Architect ist eine grafische IDE für die Entwicklung von SystemC-Modellen und deren Integration zu einer virtuellen Plattform, die anschließend für die Software-Entwicklung verwendet werden kann. Die zentrale grafische Benutzeroberfläche (GUI) ist der Platform-Creator. Dort können existierende SystemC-Modelle importiert und grafisch zu einem kompletten System verbunden werden. Importiert werden können dabei Modelle aus der CoWare-Bibliothek, aus benutzereigenen Bibliotheken sowie (mit oder ohne Unterstützung anderer CoWare-Tools) selbst erstellte Modelle und Modelle anderer Hersteller, jeweils die Real-Interoperabilität auf Transaktionsebene vorausgesetzt. SystemC-Modelle können mit der Eclipse-basierten SystemC-IDE (SCIDE) neu erstellt oder bearbeitet werden. Ein weiteres Tool als Bestandteil des Platform-Architects ist der SystemC-Component-Wizard (SCWizard), ein grafisches Tool zum Erstellen von Modellen. Der SystemC-Explorer ist eine grafische Debugging-Umgebung, die umfangreiche Analysemöglichkeiten für SystemC und TLM zur Verfügung stellt. Architektur-Exploration und Performance-Analyse sollen mit dem Platform-Architect ebenfalls möglich sein.

Die Bestandteile des Platform-Architect unterstützen eine herstellerspezifische SystemC-Bibliothek, die *SystemC-Modeling-Library (SCML)* [CoWa09]. Diese stellt verschiedene Modellierungsobjekte zur Verfügung, die den Entwickler bei der Erstellung von Modellen unterstützen und in den CoWare-Tools für die Analyse und andere Aufgaben integriert sind. Die SCML ist nicht auf die Verwendung im CoWare-Tool beschränkt, sondern ist auch als kostenloser Download [CoWa10b]^i für die Verwendung in anderen Entwicklungsumgebungen verfügbar. Es gibt beispielsweise Clock-Objekte, TLM-Adapter, Speicher- und Registerobjekte, die überwiegend der Erstellung von Real-Modellen dienen. Für die Konfiguration sind vor allem die Property-Objekte von Interesse, die nach Abschnitt 2.2 der Erstellung von Meta-Modellen zuzuschreiben sind.

[7] Die für diese Arbeit verwendete CoWare-Platform-Architect-Version ist v2009.1.1.

SCML-Properties sind typisierte Objekte, die im Modell-Quellcode instanziiert werden können und über eine Datenbank, SCML-Property-Registry genannt, einen Initialwert erhalten können. Properties können entweder manuell (z.B. mit der SCIDE) eingebunden werden oder im Verlauf der grafisch geführten Generierung von Modellen mit dem SCWizard automatisch eingefügt werden[8]. Nach einem anschließenden Import des Modells in den Platform-Creator können die Properties unter Angabe ihres Datentyps entweder mit grafischer Unterstützung oder mit einem Tcl[9]-Befehl beim Tool registriert werden. Während der Simulation im Platform-Creator erhalten die Property-Objekte ihre per GUI oder Tcl-Skript zugewiesenen Werte während ihrer Konstruktion. Die Objekte können eindeutig über einen Namen identifiziert werden, der sich aus einem (Modul-)lokalen benutzergewählten Namen und einem automatisch ermittelten, von der SystemC-Hierarchie abhängigen Präfix zusammensetzen. Die Tabelle 3.3 auf Seite 36 listet die möglichen Datentypen auf.

Es folgen eine Analyse der internen Abläufe des quelloffenen SCML-Kits und eine Diskussion der dadurch entstehenden Möglichkeiten und Einschränkungen für die Modell-Konfiguration.

Property-Objekte dürfen nur in SystemC-Modulen verwendet werden. Sie verkörpern Datentypen und können direkt wie diese eingesetzt werden. Das Hinzufügen neuer Property-Typen ist nicht möglich, da sich spezielle Funktionen für die existierenden Typen in verschiedenen internen Klassen befinden, auch im Property-Server-Interface, das nicht geändert werden kann (z.B. `getIntProperty`, `getStringProperty` etc.). Während des Konstruierens des Objekts (also noch vor Ausführung des Modul-Konstruktor-Bodys, wenn es sich bei der Property um einen Klassenmember handelt) wird der Property ihr Wert automatisch aus der Property-Registry zugewiesen: Dabei wird der Wert der Membervariablen `mValue` aus dem global verfügbaren Singleton[10] Registry-Objekt `scml_property_registry::inst()` geholt. Dort wird der Wert anhand des übergebenen Namens aus dem globalen Property-Server `mCustomPropServer` ausgelesen. Wenn es keinen vorher festgelegten Initialwert in der Registry gibt, wird ein im Property-Konstruktor optional übergebener Wert (Defaultwert) angewendet. Der Initialwert hat also Vorrang vor dem Defaultwert.

Der Property-Server ist Teil des Platform-Creators und liest die Initialwerte aus einer XML-Datei, ist aber nicht quelloffen. Alternativ kann der Benutzer einen eigenen Property-Server betreiben, indem er das Interface `scml_property_server_if` implementiert.

Da die Konfigurationswerte aus einer Datei gelesen werden, und nur beim Konstruieren der Property-Objekte angewendet werden, kann mit SCML lediglich eine einmalige Konfiguration pro Property vorgenommen werden. Folglich gibt es auch keinen Mechanismus, der über Wertänderungen informiert. Modulintern können Properties dagegen beliebig verwendet, also

[8] Der SCWizard macht die Properties bei der Generierung des Modell-Quellcodes automatisch auch zu Konstruktor-Parametern des Moduls, was für deren einwandfreie Funktion aber überflüssig ist.

[9] Tcl ist eine Skriptsprache, die speziell auf die Steuerung von Programmen, GUIs u.ä. ausgerichtet ist.

[10] Ein Singleton (Einzelstück) ist ein Entwurfsmuster, das dafür sorgt, dass von der betreffenden Klasse nur ein einziges Objekt erzeugt werden kann, auf das in der Regel über eine globale Funktion zugegriffen werden kann (vgl. [GHJV94]).

auch geändert werden. Die Rangfolge der Wertzuweisungen ist nach Zeitpunkten festgelegt, d.h. der jeweils zuletzt zugewiesene Wert ist der Wert der Property.

Die SCML-*Modell-API* besteht aus den einfach gehaltenen SCML-Properties in der Klasse `scml_property<Typ>`. Der Template-Typ legt den Datentyp fest, den die Property repräsentieren und speichern soll. Es werden nur die Datentypen `int`, `unsigned int`, `double`, `bool` und `std::string` unterstützt. Listing 2.5 ist ein Beispiel für die Verwendung von Properties in einem SystemC-Modul. Das Erzeugen von konfigurierbaren Properties wird entweder manuell im Quellcode oder im Zuge der Quellcode-Generierung mit dem Tool SCWizard vorgenommen.

```
class mymodule : public sc_module
{
public:
  SC_HAS_PROCESS(mymodule);

  mymodule(sc_module_name name)
  : sc_module(name),
    m_int_prop("m_int_prop"), // Parameter ohne Defaultwert
    m_bool_prop("m_bool_prop", true) // Parameter mit Defaultwert
  {
    SC_THREAD(my_thread);
  }
  scml_property<int> m_int_prop;
  scml_property<bool> m_bool_prop;

  void my_thread () {
    cout << name() << " property m_int_prop " << m_int_prop << endl;
    cout << name() << " property m_bool_prop " << m_bool_prop << endl;
  }
};
```

Listing 2.5: Beispiel-Modul mit SCML-Properties

Die wichtigsten Elemente der *Tool-API* für die Konfiguration mit SCML sind die folgenden:

- Die Initialwerte der Properties können im Tool Platform-Creator vor Beginn der Simulation über verschiedene Schnittstellen gesetzt werden, die die Werte dann statisch über eine XML-Datei für die Simulation bereitstellen:
  - grafisch oder
  - über eine Kommandozeile per Tcl-Skript mit den beiden Befehlen
    `create_user_parameter mymodule m_str_prop string`
    (zum Registrieren der Property) und

`set_user_parameter_property mymodule m_str_prop`
`default_value "some value"` (zum Setzen eines Initialwertes).

- Alternativ können die Initialwerte über eine benutzerspezifische Implementierung des SCML-Server-Interfaces `scml_property_server_if` zur Verfügung gestellt werden, der bei der Property-Registry `scml_property_registry` registriert wird.

- Das Auslesen der initialen Property-Werte kann durch in die Simulation integrierte Tools ebenfalls über die Property-Registry geschehen. Die Funktionen dafür sind die folgenden:

```
long long getIntProperty(const std::string & name);
unsigned long long getUIntProperty(const std::string & name);
bool getBoolProperty(const std::string & name);
std::string getStringProperty(const std::string & name);
double getDoubleProperty(const std::string & name);
```

Zusammenfassung

SCML bietet konfigurierbare Objekte, die mit Templates einige vorgegebene Datentypen repräsentieren und transparent wie diese Datentypen verwendet werden können (Class-Wrapper). Sie ermöglicht den Zugriff auf ein globales Verzeichnis aller Objekte und kann die Konfiguration in eine Datei exportieren. Das eine Property besitzende Modell kann Defaultwerte setzen, die potentiell von Initialwerten in der Datenbank überschrieben werden. Die Rangfolge der Wertzuweisungen richtet sich nach dem Zeitpunkt, der zuletzt geschriebene Wert hat Vorrang. Benachrichtigungen über Wertänderungen sind nicht vorgesehen.

2.6.4. ARM CASI

ARM bietet im Rahmen der *RealView-ESL-API* einen Konfigurationsmechanismus an. Die RealView-ESL-API wird von der Entwicklungsumgebung SoC-Designer verwendet. Sie enthält das *Cycle Accurate Simulation Interface (CASI)*[11], das einen Konfigurationsmechanismus mit Konfigurations-Interfaces definiert und dessen Dokumentation [ARM06] und Implementierung für das Review verwendet wurden. Der SoC-Designer wurde von ARM entwickelt und wird nun von Carbon Design Systems weitergeführt [ARM08][i].

Das CASI sieht vor, die generische Programmierung von Modellen, Ports und Channels für eine zyklenbasierte Simulation zu ermöglichen. Dazu gehören Konfigurations-Parameter, die zur Laufzeit gesetzt werden können.

[11] Die Aussagen des Reviews beziehen sich auf die ausführlich untersuchte CASI Version 1.1.0, für die eine quelloffene Implementierung existiert. Das Konfigurations-Interface der aktuelleren Version 2.0 [ARM07] unterscheidet sich laut der Dokumentation nicht von der untersuchten Version, es gibt allerdings keine frei verfügbare Implementierung.

Die CASI-Konfiguration bietet sehr einfache Parameter an, deren Wert ausschließlich durch einen String repräsentiert wird. Parameter können vom besitzenden Modul erzeugt und anschließend über eine C++-Schnittstelle gesetzt und gelesen werden. Parameter werden über einen (lokalen) Namen identifiziert, der Wert wird über diese Schnittstelle ausschließlich als String dargestellt. Es gibt keine zentrale Parameterverwaltung, deswegen lässt sich eine Parameterliste auch jeweils nur für ein Modul abrufen.

Das konfigurierbare Modul muss von der Klasse `casi_module` ableiten, wo die Konfigurations-Funktionen als (eingeschränkt funktionsfähige) Implementierung vorhanden sind. Sie lassen sich vom Modul überschreiben, wenn die Möglichkeit erweiterter Merkmale genutzt werden soll. Ein Parameter kann beim Erzeugen Eigenschaften erhalten (vom Typ `CASIParameterProperties`), die von der Standard-Implementierung gespeichert, aber sonst ignoriert werden.

Optional anwendbare Eigenschaften von Parametern sind:

- Datentyp (`type`): undefined, string, bool, value, symbol,
- Bereiche (`range`): eine Menge von (nicht näher definierten) Wertebereichen,
- Laufzeitparameter (`is_runtime`): ein (nicht näher definierter) Laufzeitparameter,
- privater Parameter (`is_private`): ein (nicht näher definierter) privater Parameter,
- schreibgeschützter Parameter (`is_readonly`): ein (nicht näher definierter) nur lesbarer Parameter,
- Symbole (`symbols`): ein Array von Zeichenketten (ohne näher definierte Bedeutung).

Die Semantik dieser Eigenschaften sowie eine Implementierung sind nicht vorgegeben.

Die Parameter sollen in der Elaborationsphase gesetzt werden, und zwar bevor die Komponenten miteinander verbunden werden. Die Rangfolge der Wertzuweisungen ist zeitlich festgelegt, d.h. der jeweils zuletzt zugewiesene Wert ist der Wert des Parameters.

Die CASI-*Modell-API* ist die bereits erwähnte Klasse `casi_module` sowie deren (protected) Funktion `void defineParameter (const std::string &key, const std::string& value, const CASIParameterProperties* prop = NULL)` bzw. eine Variante dieser Funktion mit Auflistung der `CASIParameterProperties`. Diese Funktionen erstellen den Parameter mit einem Defaultwert. Das Erstellen ist notwendig, damit ein Parameter von anderen Modulen oder Tools gesetzt werden kann. Die Anwendung der optionalen Eigenschaften ist nicht vorgesehen und wird in der vorhandenen Implementierung ereignislos gespeichert. Eine eigene Implementierung könnte die Eigenschaften auswerten. Ein Modul kann auf die eigenen, von der Basisklasse verwalteten Parameter über die Funktionen der Tool-API zugreifen. Wenn ein Modul die Funktionen überschreibt und die Parameterverwaltung selbst übernimmt, hat es direkten Zugriff auf die Parameter und könnte sie z.B. sogar in anderen Datentypen speichern und bei Bedarf eine String-Konvertierung für die API-Aufrufe vornehmen.

Die CASI-*Tool-API* ermöglicht den Zugriff auf die Parameter von außerhalb des besitzenden Moduls mit den folgenden Funktionen:

- `void setParameter (const std::string& key, const std::string& value)` setzt den Wert (als String) eines (existierenden) Parameters.
- `std::string getParameter (const std::string& key)` liefert den Wert eines Parameters (als String).
- `const CASIParameterMapIF* getParameterList()` liefert eine CASI-Datenstruktur mit allen Parametern des Modules, inklusive den String-Werten und optionalen Eigenschaften.
- `bool testParameter(const std::string& key, const std::string& value, std::string& error_message)` soll vermutlich[12] zurückliefern, ob der Wert wie vorgegeben gesetzt ist und sonst die Fehlernachricht ausgeben. (Diese Funktion muss in jedem Fall überschrieben werden, wenn sie verwendet werden soll.)

Da weder die Semantik noch die Implementierung der optionalen Parametereigenschaften vorgegeben ist, entscheidet die jeweilige Implementierung darüber. Alternativ können diese Eigenschaften als rein informativ gelten, also keinerlei Einfluss auf das Verhalten haben.

Zusammenfassung

ARM CASI verwendet sehr einfach gehaltene Parameter, die über ein Konfigurations-Interface verwendet werden. Die vorgehaltene Implementierung ist ebenfalls einfach und weitergehende Eigenschaften müssen durch benutzereigene Implementierung unterstützt werden. Die API besteht aus Funktionen und ist einfach und übersichtlich, der Modul-interne Zugriff auf die Parameter dagegen aufwändig. Problematisch ist beispielsweise die Funktion `testParameter`, die (undokumentiert) immer wahr zurückgibt, solange sie nicht vom Modul überschrieben wird. Die Parameter sind verteilt gespeichert und werden nicht in einem zentralen Verzeichnis verwaltet. Wertänderungen durch ein Tool oder den Besitzer sind auch während der Simulationsphase möglich, wobei der zeitlich letzte Wert übernommen wird. Benachrichtigungen über Wertänderungen sind nicht vorgesehen.

2.6.5. Synopsys CCSS-Parameter

Eine Entwicklungsumgebung, die SystemC-Konfiguration anbietet, ist der Innovator von Synopsys [Syno08][i]. Es handelt sich um eine IDE für die Erstellung von virtuellen Plattformen mit SystemC. Außerdem soll das Programm vom Entwickler auch verwendet werden können, um die erstellten virtuellen Plattformen zu analysieren und zu verifizieren. Nachdem

[12] Über die Aufgaben einiger Elemente können nur Vermutungen angestellt werden, da sie undokumentiert sind.

die virtuelle Plattform zusammengestellt und verifiziert ist, soll die Plattform vom Innovator für Software-Entwickler zur Verfügung gestellt werden können. Eine virtuelle Plattform ist in diesem Fall ein voll funktionsfähiges (Software-)Modell eines Hardware-Designs. Auf einer solchen virtuellen Plattform kann laut Synopsys Software entwickelt werden, die später unverändert auf der Hardware läuft.

Synopsys stellt die DesignWare-System-Level-Library zur Verfügung, die vorgegebene Modelle enthält. Sie können in der Plattform mit beliebigen SystemC-Modellen kombiniert werden. Eine grafische Oberfläche, die die simulierte Plattform-Oberfläche widerspiegelt, sowie eine Verknüpfung zu Komponenten des Host-PCs sind möglich.

Für die Konfiguration dieser Modelle stehen grafisch angezeigte und zu bearbeitende Parameter zur Verfügung.

Für die Konfiguration der Modelle stellt der Innovator sogenannte *CCSS-Parameter* (Klasse `ccss_param`) zur Verfügung[13]. Die wesentlichen *Tool-API*-Merkmale sind:

- Grafische Anzeige von CCSS-Parametern und Defaultwerten vor der Simulation,
- Grafisches Ändern von CCSS-Parameterwerten ausschließlich vor der Simulation,
- Zugriff auf CCSS-Parameter innerhalb SystemCs während der Simulation.

CCSS-Parameter sind Konfigurationsobjekte, die im SystemC-Code erstellt werden können und die *Modell-API* repräsentieren. Konzeptionell sind sie Paare bestehend aus einem Namen (Typ String) und einem Wert (vom jeweiligen Datentyp). Parameter können im Quelltext ausschließlich als Klassenmember verwendet werden. Der Innovator zeigt die Parameter (Name und Defaultwert) eines Modells grafisch an. Der Innovator verfolgt ein Konzept von lokalen Parametern: Hierarchisches Setzen von Parametern ist innerhalb des Modells über eine spezielle Funktion möglich. Nur ein Parent-Modul kann Parameter eines Child-Moduls setzen. Die Innovator Benutzeroberfläche zeigt nur Parameter der obersten Instanzierungsebene an, nicht von Untermodulen.

Defaultwerte müssen im Modul-Konstruktor über eine bestimmte Funktion gesetzt werden. Wertänderungen sind auch zur Laufzeit möglich, allerdings nur, wenn sie vom Modell selbst durchgeführt werden, und die Änderungen werden von der GUI nicht angezeigt (auch nicht über z.B. die Konsole). Initialwerte können von der GUI vor dem Starten der Simulation geändert werden und überschreiben die im Quellcode gegebenen Defaultwerte. Die Namensgebung funktioniert hierarchisch, wobei der lokale Name automatisch auf den Variablennamen gesetzt wird.

Es werden vielfältige Datentypen unterstützt, konkret sind das C++- und SystemC-Datentypen, sowie Synopsys-spezifische und benutzerdefinierte.

[13] Die Buchstaben CCSS werden im Innovator nicht mit einer Bedeutung versehen, vermutlich stammen sie vom Synopsys CoCentric-System-Studio.

CCSS-Parameter unterstützen (anhand der vorhandenen Interfacemethoden erkennbar[14]) Benachrichtigungen (über SystemC-Event-Notifikationen) bei Wertänderungen. Direkte bzw. sofortige Callbacks werden nicht unterstützt.

Der Innovator parst den Modell-Quellcode und sucht dabei nach Parametern. Sowohl der Name als auch der Defaultwert werden über Quelltextparsing ermittelt. Die Hilfetext-Anleitung gibt sehr genaue Anweisungen, wie die Parameter im Code erstellt und mit einem Defaultwert belegt werden können.

Leider hat die verwendete Innovator-Version[15] einen Bug: Die im Quellcode festgelegten Defaultwerte von Parametern, auch die der originalen Synopsys Parameter, werden in der GUI nicht angezeigt. Ungeachtet dessen werden sie richtig angewendet und simuliert.

Zusammenfassung

CCSS-Parameter sind Class-Wrapper-Objekte, die vielfältige Datentypen unterstützen. Parameter und -werte können im GUI für das jeweils ausgewählte Modul angezeigt und Initialwerte können vor dem Beginn der Simulation gesetzt werden. Dieser Wert überschreibt einen im Modell – potentiell im Subsystem vom übergeordneten Modul – festgelegten Defaultwert. Während der Programmphase vom Besitzer geänderte Werte können vom Tool nicht abgefragt und aktualisiert werden. Abgesehen von der Präferenz der Initialwerte vor den Defaultwerten ist der jeweils zeitlich zuletzt gesetzte Wert der Parameterwert. Das Modell kann sich mit SystemC-Events über Änderungen informieren lassen.

2.6.6. Open-Verification-Methodology (OVM)

Die *Open-Verification-Methodology (OVM)* ist eine auf Verifikation ausgerichtete, herstellerunabhängige Open-Source-Methodik für alle IEEE-Verifikationssprachen, die gemeinsam von Cadence Design Systems und Mentor Graphics eingeführt wurde (vgl. [Cade10]). OVM liegt für die IEEE-Sprachen SystemVerilog [IEEE09], *e* [IEEE08] und SystemC [IEEE06] vor. Viele Konzepte stammen aus der Verifikationssprache *e*. Die hier untersuchte SystemC-Variante [Cade09] wird von Cadence gepflegt und bietet eine für SystemC geeignete Untermenge der OVM-Merkmale an. OVM soll eine Mischung von unterschiedlichen Sprachen in einer Testbench unterstützen und bietet zu diesem Zweck beispielsweise Kommunikationskanäle an. Ein wesentliches Ziel ist die Wiederverwendbarkeit von Code, wie es auch das Interoperabilitätsziel dieser Arbeit anstrebt.

Dieses Review hat seinen Fokus auf dem Konfigurationsmechanismus, der in Teilen unabhängig von der Sprache ist. Da in dieser Arbeit die Sprache SystemC verwendet ist, wird der SystemC-Konfigurationsmechanismus [Cade09] untersucht. Die SystemC-Realisierung kann im Bereich „OVM World Contributions“ der OVM-Webseite frei heruntergeladen [Cade10][i] werden.

[14] Event-Notifikationen haben in Tests nicht funktioniert.
[15] Untersucht wurde die Innovator Version 2008.12-SP2, Build 20081202009

OVM erlaubt das Setzen und Lesen von konfigurierbaren Parametern über Interface-Methoden. Die Parameter können während der Programmphase gesetzt werden, bevor die besitzende Komponente instanziiert wird. Es werden drei verschiedene Parameter-Datentypen unterstützt: Zahlen „int“ verschiedenen Typs (d.h. `bool`, `char`, `sc_dt::uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `sc_dt::int64`, `sc_dt::uint64`), Zeichenketten „string“ und Objekte „ovm_object“ (eine beliebige Komposition aus primitiven Datentypen). Für diese drei Typen gibt es jeweils eine Funktion `set_config_[int|string|object]` zum Setzen von Werten und eine entsprechende Funktion `get_config_[int|string|object]` zum Lesen des Wertes. Die Datenbank ist verteilt, d.h. die Werte werden jeweils dort gespeichert, wo der Funktionsaufruf getätigt wird. Beim Abrufen des Wertes wird eine hierarchische Suche durchgeführt, beginnend bei der höchsten Komponente, herunter bis zur Position, wo der Abruf getätigt wurde. Der erste Eintrag, der auf den abgefragten Parameter passt, wird als Wert zurückgegeben.

Das führt dazu, dass hierarchisch höhere Komponenten die Parameterwerte der niedrigeren überschreiben können. Des Weiteren bedeutet dieses Verfahren bei jedem Abruf des Parameterwertes eine relativ aufwändige Suche nach dem Wert.

Es ist in OVM explizit empfohlen, sämtliche Parameter zu konfigurieren, bevor das Subsystem erzeugt wird, das die Parameter verwendet (vgl. [Cade09, S. 8]). Folglich ist es nicht erwünscht, dass vom Besitzermodul bereits gelesene Parameterwerte wieder geändert werden.

Ein SystemC-Modul wird konfigurierbar, indem es von der Klasse `ovm_component` ableitet. Die Klasse wird als Teil der Bibliothek bereitgestellt, d.h. die Konfiguration ist dort implementiert und vorgegeben und braucht nicht vom Benutzer selbst implementiert zu werden.

Das Modell sollte eigene Objekte mit entsprechenden Datentypen verwalten. Eine Verbindung zum Konfigurationsmechansimus wird ausschließlich durch die aktiven Funktionsaufrufe des Modells in der Komponentenklasse (`ovm_component`) hergestellt, von dem das `sc_module` des Benutzers ableitet. Dadurch handelt es sich um einen reinen Poll-Mechanismus, in dem das Modell selbst Änderungen der Parameter abfragen und mitteilen muss – das gleiche gilt für ein Tool. Es gibt keinen Mechanismus, der über Wertänderungen informiert.

Für die von OVM angebotene Schnittstelle kann kaum zwischen *Modell-API* und *Tool-API* unterschieden werden, da die gleichen Funktionsaufrufe sowohl vom Modell als auch von einem potentiell übergeordneten Tool verwendet werden. Die folgenden hier verkürzt dargestellten Funktionen bietet die Klasse `ovm_component` für das Setzen und Lesen von Parametern an:

- `template <typename T>`
  `void set_config_int(string& instname, string& field, T& val)`
  setzt den Wert eines Parameters vom Typ Zahl.

- `void set_config_string(string& instname, string& field, string& val)`
  setzt den Wert eines Parameters vom Typ String.

- `void set_config_object(string& instname, string& field, ovm_object* val, bool clone)` setzt den Wert eines Parameters vom Typ Objekt.

- `template <class T> bool get_config_int(string& field, T& val)`
  liefert den Wert eines Parameters vom Typ Zahl.

- `bool get_config_string(string& field, string& val)`
  liefert den Wert eines Parameters vom Typ String.

- `bool get_config_object(string& field, ovm_object*& val, bool clone)`
  liefert den Wert eines Parameters vom Typ Objekt.

Diese Konfigurationsfunktionen erhalten als Funktionsparameter den Instanznamen des Moduls, das den Parameter besitzt und den Namen des gewünschten Parameters[16]. Funktionen, die einen Wert setzen, bekommen diesen Wert als Funktionsparameter übergeben. Funktionen, die einen Wert lesen, bekommen eine Referenz auf das Ziel des Lesevorganges als Funktionsparameter. Die Funktionen für das Setzen und Lesen von Zahlenwerten sind Template-Funktionen, die die oben genannten Datentypen verarbeiten können. Die Voraussetzung für die dort verwendbaren Zahlentypen ist, dass sie in einem 4096-Bit breiten `sc_lv<4096>` gespeichert werden können.

Beim Setzen von Werten können in den Instanznamen Wildcards verwendet werden. Beim Abruf eines Wertes wird top-down überprüft, ob ein Eintrag auf den gesuchten Parameternamen passt.

Die Komponente, die den Parameter besitzt, registriert diesen mit einem Aufruf einer der `get`-Funktionen und kann mit der entsprechenden `set`-Funktion einen Defaultwert angeben, der von dem hierarchisch höchsten Wert überschrieben wird.

Abbildung 2.6 ist ein Beispiel, das die hierarchische Rangfolge und die Verwendung von Wildcards zeigt. Die linke Seite zeigt die Klassenstruktur mit Code-Beispielen[17], die rechte Seite zeigt die resultierende Objektstruktur. Im Modul `ovm_component_A` wird ein Wildcard verwendet, um in allen Untermodulen von `childB` den Parameter `m_int` mit dem Wert `100` zu setzen. Die Komponente `ovm_component_B` schreibt den gleichen Parameter, ihre Werte (`10` und `11`) werden aber von der hierarchisch höheren Komponente überschrieben. In den beiden Objekten `childC1` und `childC2` ist zu sehen, dass der Wert `100` übernommen wird. Es wird deutlich, dass der Zeitpunkt der jeweiligen Konfigurationsaufrufe nicht die Rangfolge der Wertzuweisung bestimmt, sondern die hierarchische Position des Aufrufs.

[16]Die untersuchte Implementierung speichert die Parameterwerte intern (in der verteilten Datenbank der Komponentenklasse) in typisierten Class-Wrapper-Objekten (bestehend aus der Basisklasse `ovm_config_item` und und den abgeleiteten Klassen `ovm_config_item_*`), die den jeweiligen Datentypen speichern.

[17]Das Konfigurieren wird in diesem Beispiel jeweils im Konstruktor durchgeführt. Dies kann auch in der OVM-`build`-Funktion stattfinden, die eine Entsprechung von `before_end_of_elaboration` ist.

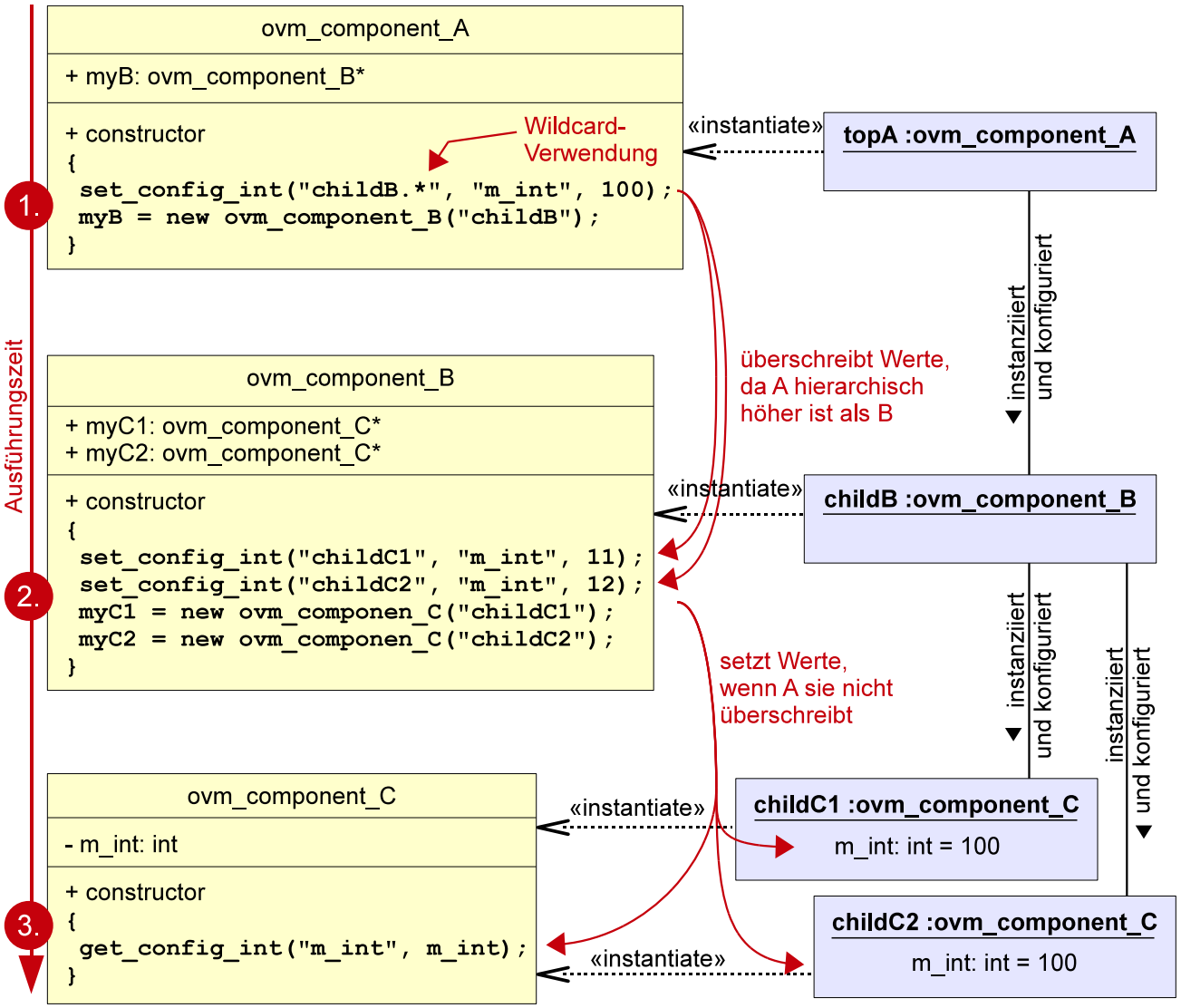


Abbildung 2.6.: OVM-Beispiel für hierarchische Reihenfolge und Wildcards

Zusammenfassung

Der OVM-Konfigurationsmechanismus bietet einfache und übersichtliche Konfigurations-Interface-Funktionen, deren Implementierung von der Bibliothek bereitgestellt wird. Eine Besonderheit ist die hierarchische (nicht zeitliche) top-down Rangfolge, in der die Konfiguration angewendet wird. Die unterstützten Datentypen sind verschiedene numerische Typen, Strings und eine Objektstruktur, die vom Benutzer gefüllt werden können. Zugriff auf ein globales Parameterverzeichnis ist nicht vorgesehen. Ein Defaultwert kann vom Besitzer beim Registrieren des Parameters gesetzt werden. Das Interface begünstigt das Anlegen einer lokalen Kopie des Parameters beim Besitzer, weswegen nur explizit über das Interface gesetzte Werte auch vom Tool ausgelesen werden können. Ein vom Tool z.B. während der Simulationsphase gesetzter Wert wird vom Besitzer nur erkannt, wenn dieser aktiv abgefragt wird. Wildcards sind beim Setzen von Parametern möglich.

2.6.7. Texas Instruments SystemPython

Texas Instruments (TI) *SystemPython* [BeAl07] schafft eine Verbindung zwischen der Skriptsprache Python und SystemC. Es handelt sich um ein von TI intern verwendetes Tool, das teilweise öffentlich verfügbar gemacht und letztendlich von GreenSocs weiterentwickelt wurde und aktuell unter dem Namen GreenScript als Open-Source verfügbar ist[18].

SystemPython erweitert Python um Threads, Events, Signale, Simulationszeit und Kommunikation zwischen Threads sowie um Nachrichtenübermittlung in die SystemC-Welt und erlaubt es, SystemPython-Objekte als SystemC-Threads beim SystemC-Kernel zu registrieren. Da die Entwicklung von Python-Code schneller und flexibler als SystemC-Code ist und leicht zu erlernen ist und Python zudem als Teil von C++ – und damit SystemPython als Teil von SystemC – eingebunden werden kann, eignet es sich für die schnelle Entwicklung von Testfällen oder Modellen.

SystemPython wird als `sc_module` in eine SystemC-Simulation eingebunden und bekommt einen Skript-Dateinamen als Parameter übergeben. Während der Elaborationsphase startet das Modul einen Python-Interpreter und kann SystemC-Threads im SystemC-Simulationskernel mit dem Aufruf `spy_create_thread` registrieren. Mit diesem Thread wird dann ein ausführbares Python-Objekt verknüpft.

Für die Konfiguration bietet SystemPython Mechanismen zur Übermittlung von Nachrichten und Parametern von SystemPython an SystemC. Zu diesem Zweck kann entweder ein globaler Satz von Parametern verwendet oder ein separater Parametersatz angelegt werden. Ein solcher Parametersatz kann mit mehreren Parametern gefüllt werden, die aus einem Identifizierer (Namen) und einem Wert bestehen.

In der *Tool-API* wird auf Python-Seite mit der Funktion `spy_parameter_set` ein Parametersatz erzeugt und einer Variablen zugewiesen (z.B. `pars = spy_parameter_set('name')`). Anschließend kann der Parametersatz mit Parametern (vermutlich Python-Typen, die eine SystemC-Entsprechung haben) gefüllt werden (z.B. `pars.image_resolution = 'vga'`). Beim Erstellen und Starten eines SystemC-Threads wird der gewünschte Parametersatz übergeben: `start_task('threadName', pars, done_event)`.

Die *Modell-API* befindet sich auf der SystemC-Seite. Dort muss der Parametersatz ausgelesen und lokal behandelt, d.h. gespeichert oder ausgewertet werden. Dynamische Änderungen der Werte zu beliebigen Zeitpunkten sind nicht vorgesehen, allerdings können Threads bzw. Tasks mit neuen Parametersätzen dynamisch gestartet werden. Die Realisierung einer Rückmeldung ist vom Benutzer implementiert denkbar über das Event (`done_event`), das in Python ausgelöst wird, wenn ein SystemC-Thread (Task) beendet ist.

Zusammenfassung

Über SystemPython liegen mir nur beschränkte Informationen vor, da es nicht frei zugänglich ist. Es ist allerdings ein interessanter Ansatz, der eine Skriptsprache mit SystemC verbindet

[18] GreenScript kann auf der Projektwebseite `http://www.greensocs.com/en/Projects/GreenScript` heruntergeladen werden

und über ein Konfigurations-Interface mehrere Parameter in einem Konfigurationsobjekt übergeben kann. Es werden Datentypen unterstützt, die zwischen Python und SystemC kompatibel sind, z.B. numerische und String-Typen. Das Setzen von Defaultwerten ist im Modell manuell möglich, genauso kann der Parameterwert intern während der Simulation geändert werden. Ein Setzen des Wertes während der Simulationsphase ist über Task-Aufrufe möglich, die einen Parametersatz enthalten; eine Rückmeldung an Python ist nur über Task-finished-Events möglich. Die Werte des jeweils zuletzt an den Task übergebenen Parametersets werden angewendet; die Rangfolge ist folglich zeitlich.

2.6.8. Intel DRF

Ein weiterer untersuchter Konfigurationsmechanismus ist der vom Intel *Device Register Framework (DRF)* [Patr07][i], einem für interne Zwecke entwickelten SystemC-Framework für die Modellierung von Registern, die auch konfiguriert werden können. Eine Weiterentwicklung ist mittlerweile unter dem Namen GreenReg bei GreenSocs frei verfügbar, siehe Abschnitt 6.8.1.

DRF ist eine Bibliothek, die einige Erweiterungen und Vereinfachungen für die Entwicklung mit SystemC zur Verfügung stellt: Die hierarchische Organisation von Modellen wird durch zusätzliche Modulklassen und Zugriffsfunktionen unterstützt. Der Entwurf, die Verbindung und Verwendung von Modulen, die über einen Bus kommunizieren, wird vereinfacht. DRF bietet Registerimplementierungen mit einfachem Zugriff (über den Bus sowie lokal innerhalb des Moduls), inklusive bitgenauem Zugriff.

Die Konfiguration von Registerwerten und beliebigen Modul-Membern ist hier von besonderem Interesse. Beliebige Datentypen sind konfigurierbar, wenn sie zwei Interfaces implementieren: Das Interface `I_dr_config` kann eine als Zeichenkette dargestellte Konfiguration entgegennehmen und die (Benutzer-)Implementierung soll die Konfiguration dann für den entsprechenden Datentypen anwenden. Der umgekehrte Weg, d.h. das Auslesen einer Konfiguration, wird vom Benutzer über die Implementierung des Interfaces `I_dr_dump` ermöglicht. Für einige Elemente von DRF (z.B. Register) ist der Konfigurationsmechanismus bereits vorhanden, weitere Datentypen können vom Benutzer hinzugefügt werden.

Das Anwenden einer Konfiguration aus einer Datei bzw. das Schreiben einer Konfiguration in eine Datei sind vorbereitet.

Die *Modell-API* der DRF-Konfiguration ist entweder die spezifische Schnittstelle des jeweils konfigurierten Objekts selbst, also beispielsweise das Registerinterface oder die benutzerspezifische Klasse, die mit der Konfiguration ausgestattet wurde. Beides ist proprietär und für den Konfigurationsmechanismus von untergeordneter Bedeutung.

Die *Tool-API* wird im Wesentlichen durch die beiden Interfaces `I_dr_config` zum Setzen und `I_dr_dump` zum Lesen repräsentiert. Das erstere wird von einer übergeordneten Instanz zur Konfiguration verwendet: Die (einzige) Funktion des Interfaces ist `parse_config(string config_str)`. Die übergebene Zeichenkette `config_str` ist der zu konfigurierende Wert und wird von der Implementierung in den erwarteten Datentyp konvertiert. Das zweite Interface

`I_dr_dump` beinhaltet die Funktion `dr_dump(dr_dump_format_e format, std:: ostream& stream, unsigned int tab_level = 0)`, welche zum Auslesen des Wertes die Konfiguration des implementierenden Objekts mit den Einstellungen der Parameter `format` und `tab_level` in den als Referenz übergebenen Stream `stream` schreibt.

Zusammenfassung

Das im Review betrachtete DRF befindet sich in der Entwicklung und bietet rudimentäre Konfiguration über einfache Konfigurations-Interfaces. Für DRF-Datentypen existieren Implementierungen, benutzereigene Datentypen können hinzugefügt werden. Für die Parameterwerte gibt es vom Besitzer festgelegte Defaultwerte, die während der Simulation geändert werden können. Die Präzedenz der geschriebenen Werte ist zeitlich. Der Parameter teilt Wertänderungen über ein Event mit.

3. Interoperabilität für die Konfiguration von SystemC-Modellen

Inhalt

Die Notwendigkeit, im ESL-Design die Wiederverwendbarkeit (Reusability) von Modellen sicherzustellen bzw. zu erhöhen sowie Architektur-Exploration mit verschiedenen Varianten des Modells vorzunehmen, erfordert einen gemeinsamen Mechanismus für die Konfiguration und Untersuchung aller Modelle.

Ein bisher für SystemC von Standards noch nicht abgedeckter Bereich der Meta-Interoperabilität (vgl. Abschnitt 2.4) ist die Modell-Konfiguration, Modell-Untersuchung und Debugschnittstelle von Modellen. Lediglich eine erste eingeschränkte Debugschnittstelle ist im TLM-2-Standard definiert (vgl. [OSCI09, S. 45]).

Interoperabilität bedeutet hier beispielsweise, dass die Konfiguration der Modelle auf standardisierte Weise vorgenommen werden kann oder eine Schnittstelle für das Untersuchen von internen Vorgängen existiert. Das würde es nicht nur erlauben, herstellerunabhängig Modelle in einer Simulation zu kombinieren, sondern diese Simulation in verschiedenen Werkzeugen verschiedener Tool-Hersteller zu verwenden, zu konfigurieren, zu untersuchen und zu debuggen.

Beispielsweise sind die folgenden Vorgänge im Verlauf einer Architektur-Exploration denkbar, die der ESL-Entwickler anwenden möchte:

- Setzen von Kommunikationseigenschaften wie Bitbreiten von Bussen[1], Frequenzen, Arbitrierungsmechanismen, Protokolloptionen, vor Beginn der Simulationsphase,
- Konfigurieren von Hardware-Core-Eigenschaften, wie Bitbreiten von Bussen und funktionale Eigenschaften, ebenfalls vor Beginn der Simulationsphase,

[1] In TLM-2.0 ist das Konfigurieren von Busbreiten nur über Template-Parameter möglich, was im Sinne dieser Arbeit keine Konfiguration ist – und ein wesentlicher Kritikpunkt an TLM-2.0 ist.

- Zugriff auf Device-Register, Speicherinhalt und andere Analysedaten während der Simulationsphase,
- Variieren von Testbench-Konfigurationen, wie Auswahl von Stimulationssequenzen, Ein- oder Ausschalten von Monitoren vor der Simulationsphase.

Ein konkreteres Beispiel ist der Entwurf eines Mobiltelefons. Es soll auf der OMAP-Plattform von Texas Instruments basieren, deren virtuelle Plattform in der Entwicklungsumgebung Synopsys DesignWare mit SystemC simuliert wird. In diesem Beispiel möchte der Entwickler zusätzliche Modelle einbinden, die aus Bibliotheken anderer Hersteller stammen: einen Signalprozessor aus der CoWare-Bibliothek und Peripherie aus ARM RealView (vgl. Abbildung 3.1). Die Real-Interoperabilität zwischen den Modellen und dem OMAP-Systembus wird mit Hilfe einer Kommunikationsverbindung über den TLM-2-Standard ermöglicht[2]. Die Meta-Interoperabilität ist dagegen nicht gewährleistet: Auf der einen Seite wird im Synopsys-Tool für die Konfiguration des OMAP-Modells der proprietäre Mechanismus CCSS verwendet, auf der anderen Seite müssen die eingebundenen Modelle (Signalprozessor und Peripherie) mit den verschiedenen proprietären Mechanismen SCML und CASI konfiguriert werden. Dieses Beispiel zeigt, dass ein Entwickler, der Modelle miteinander kombinieren möchte, die mit IDEs verschiedener Hersteller erstellt wurden, in der heute herrschenden Situation verschiedene Tools für die Konfiguration verwenden muss. Das ist arbeitsintensiv und fehleranfällig, falls es überhaupt möglich ist.

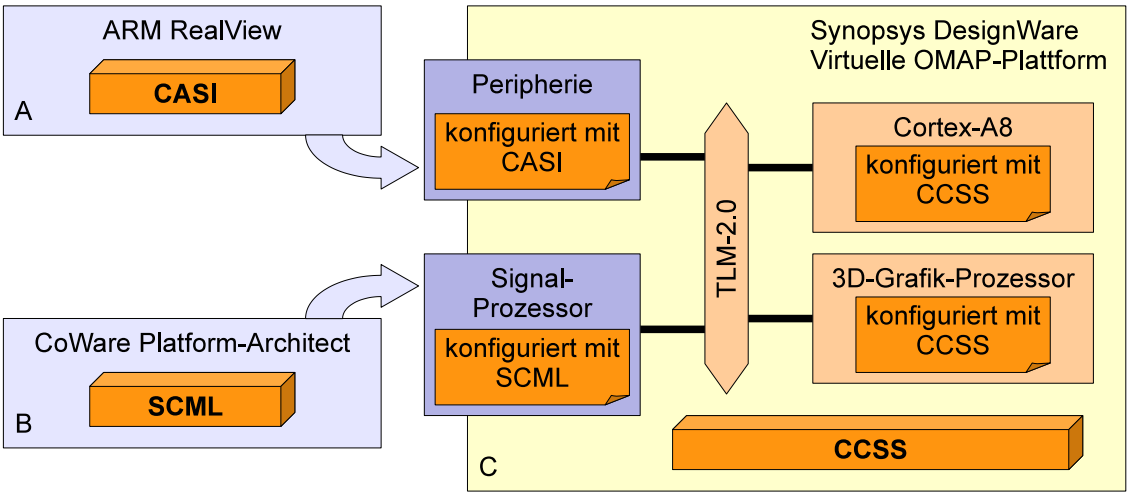


Abbildung 3.1.: Fehlende Meta-Interoperabilität

Diese Arbeit beschäftigt sich damit, diesem Problem der Meta-Interoperabilität mit einer effizienten Lösung zu begegnen.

Grundsätzlich sind zwei Wege denkbar, Meta-Interoperabilität herzustellen: Der eine Weg ist ein Standard, der dafür sorgt, dass die Modelle und Werkzeuge zueinander passen und austauschbar sind. So lange allerdings kein solcher Standard existiert, kann die Interoperabilität nur über den zweiten Weg hergestellt werden: Adapter wandeln die unterschiedlichen

[2]Davon wird hier ausgegangen, da die Real-Interoperabilität nicht im Fokus dieser Arbeit liegt.

existierenden Ansätze ineinander um (vgl. Abschnitt 3.3). Abbildung 3.2 zeigt ein grundlegendes Szenario mit dem Adapter-Ansatz. Die verschiedenen Modelle können gemeinsam über eine Adapter-Middleware mit einem der proprietären oder dem zusammen mit der Middleware entwickelten universellen Tool konfiguriert werden.

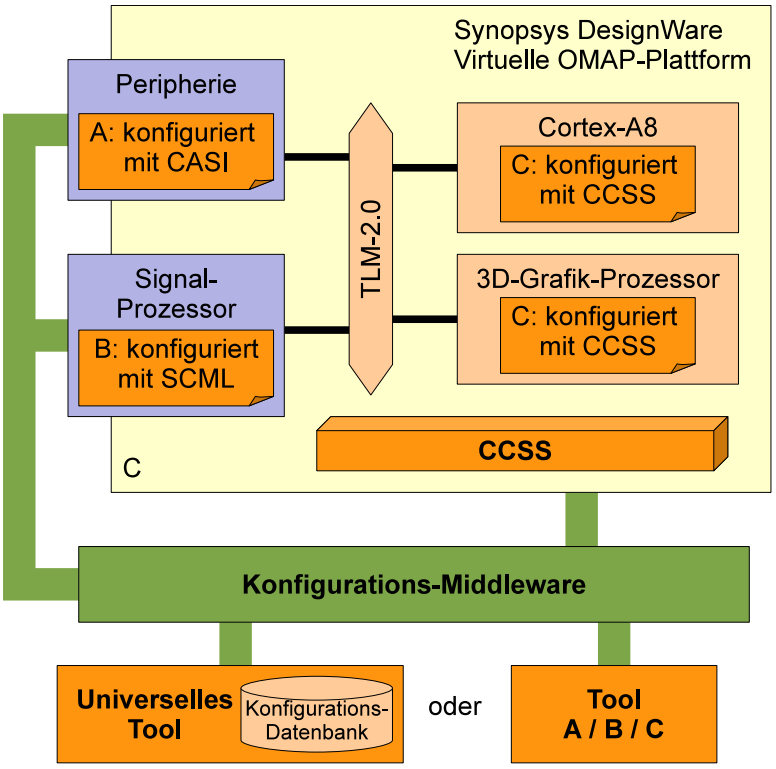


Abbildung 3.2.: Meta-Interoperabilität mit Adaptern

Diese Arbeit beschäftigt sich mit beiden Wegen zum Erreichen von Meta-Interoperabilität. Einerseits wird ein Werkzeug entworfen, welches die Entwicklung von Adaptern zwischen verschiedenen Entwicklungsprogrammen und Modellen begünstigt und einige bereitstellt. Andererseits fließen die Ergebnisse dieser Arbeit in die Standardisierung ein, mit der die OSCI in der Configuration, Control & Inspection Working Group (CCI WG) begonnen hat und sich dabei mit der Meta-Interoperabilität beschäftigt.

3.1. Analyse der existierenden Mechanismen

Von den im Abschnitt 2.6 untersuchten Mechanismen sind insbesondere die industriell verwendeten von Bedeutung für den hier entwickelten Mechanismus. Tabelle 3.3 stellt deren Konfigurationsmerkmale gegenüber. Die erste Spalte listet die als wichtig identifizierten Merkmale auf. Die letzte Spalte listet die Anforderungen an den in dieser Arbeit vorgestellten universellen Mechanismus auf, die in Abschnitt 3.2 detailliert aufgegriffen werden.

	Merkmal	SCML [CoWa09]	CASI [ARM06]	CCSS-Parameter [Syno08][i]	OVM [Cade10]	SystemPython [BeAl07]	DRF [Patr07][i]	Universeller Konfig.-Mech.
F1	Konfigurations-ansatz	Class-Wrapper	Konfigurations-Interface	Class-Wrapper	Konfigurations-interface	Konfigurations-interface	Class-Wrapper	beide, intern: Class-Wrapper
F2	Datentypen	int, unsigned int, double, bool, std::string	std::string	C++, SystemC, proprietäre, Benutzer	C++ und SystemC numerische Typen, Strings, Objekte	Python-Typen, z.B. Strings und numerische Formate	proprietäre, z.B. Register, Benutzer	PODs, SystemC, Benutzer (Interface-implementierende)
F3	Benutzerdefinierte Datentypen	-	(✓) (nur intern)	-	✓ (`ovm_object`)	(unbekannt)	✓	✓ (Template-spezialisiert)
F4	Dateiausgabe	XML-Datei	-	-	-	-	Text-Datei	Text-Datei u.a.
F5	Globales Verzeichnis	✓	(✓) verteilt	(✓) (verteilt, nur im GUI)	-	-	✓	✓
F6	Defaultwert im Modell	✓ (Konstruktor)	✓	✓	✓	Benutzer-implementiert möglich	✓	✓ (Konstruktor)
F7	Initialwert im Tool	✓	✓ (Tool-API)	✓	✓ (Tool-API)	(✓)	✓	✓
F8	Wert v. Tool lesbar während SimPh	✓	✓	-	nur initialer Wert; nicht empfohlen: expliziter Wert	✓	✓	✓
F9	Wertänderung (durch Besitzer) während SimPh	✓	✓ (vom Benutzer implementiert)	✓	(✓) nicht empfohlen	✓ (vom Benutzer implementiert)	✓ (vom Benutzer implementiert)	✓
F10	Wertänderung (durch Tool) während SimPh	-	✓	-	(✓) nicht empfohlen	✓	-	✓
F11	Benachrichtigung bei Wertänderung an Benutzer	-	-	✓ (Event)	-	✓ (Task-Aufruf)	✓ (Event)	✓ (Registrierter Callback)
F12	Benachrichtigung bei Wertänderung an Tool	-	-	-	-	Task-finished-Event	-	✓ (Tool-API registriert Callback)
F13	Param.-Zugriff mit Wildcards/RegExp	-	-	-	✓	-	-	✓
F14	Rangfolge Wertzuweisung	zeitlich	zeitlich	zeitlich	hierarchisch	zeitlich	zeitlich	zeitlich (hierarchisch nachbildbar)

Tabelle 3.3.: Merkmale (F1-F14) der untersuchten Konfigurationsmechanismen, Begriffserläuterungen siehe Abschnitt 3.2

Die meisten Merkmale ergänzen einander, schließen sich also nicht gegenseitig aus und sind somit relativ *unabhängig* voneinander und gut vereinbar. Das betrifft den Umfang der unterstützten Datentypen (F2), die Unterstützung benutzerdefinierter Datentypen (F3), die Dateiausgabe (F4), das globale Verzeichnis (F5), sowie die verschiedenen Arten der Wertzugriffe (F6 bis F10). Auch die Benachrichtigungen bei Wertänderungen (F11 und F12) und die Parameterzugriffe mit Wildcards oder regulären Ausdrücken (F13) schließen kein anderes Merkmal aus. Diese Merkmale gehen kombiniert in die Anforderungen an den universellen Konfigurationsmechanismus ein (Abschnitt 3.2), der als eines der Hauptziele dieser Arbeit entwickelt werden soll.

Zwei Merkmalspalten in der Tabelle nehmen allerdings eine Einordnung vor, deren Alternativen sich auf den ersten Blick *widersprechen*: Das sind der Konfigurationsansatz (F1), das heißt die grundlegende Art und Weise des Interfaces und die Rangfolge der Wertzuweisung (F14). Trotzdem sollen auch diese beiden sich jeweils widersprechenden Merkmale in einem universellen Konfigurationsmechanismus unterstützt werden. Sie werden deswegen in den Unterabschnitten 3.1.2 und 3.1.3 gesondert diskutiert.

3.1.1. Regelmäßigkeiten

Als Ergebnis der Reviews der verschiedenen Konfigurationsmechanismen werden in diesem Abschnitt die Gemeinsamkeiten bzw. Regelmäßigkeiten dieser Mechanismen untersucht.

Eine besonders wichtige Gemeinsamkeit in allen Konfigurationsmechanismen ist, dass für die Beschreibung einer konfigurierbaren Eigenschaft ein Paar von einem Identifizierer (z.B. einem Namen) und einem Wert verwendet wird. Eine solche Konfigurationseigenschaft, die einen variablen Wert annehmen kann, wird **Modell-Parameter** genannt.

Zugriffe auf Modell-Parameter lassen sich in zwei Gruppen einteilen: Ein **manipulativer Wertzugriff** bewirkt eine Änderung des Wertes oder eines sonstigen nach außen relevanten Zustands. Ein manipulativer Wertzugriff ist typischerweise schreibend. Dagegen hat ein **wertneutraler Zugriff** keine entsprechende Auswirkung. Lesende Zugriffe auf den Wert oder den Zustand sind immer wertneutrale Zugriffe, solange sie keinen manipulativen Seiteneffekt haben. Aber auch (schreibende) organisatorische Zugriffe wie das Registrieren von Benachrichtigungen und Ähnliches sind wertneutral, wenn sie den nach außen sichtbaren Zustand des Modell-Parameters nicht verändern, obwohl sie aus programmiertechnischer Sicht das Objekt verändern.

Eine **Benachrichtigung** (Notification) ist ein Mechanismus, der einen Beobachter, also beispielsweise ein Meta-Modell oder Tool, über eine bevorstehende oder abgeschlossene Zustandsänderung eines Modell-Parameters oder einen Zugriff auf einen solchen in Kenntnis setzt. Das Registrieren oder Abrufen einer Benachrichtigung ist – falls das für den entsprechenden Modell-Parameter vorgesehen ist – ein Beispiel für einen wertneutralen Zugriff.

Zwei schon im Stand der Technik aufgetretene Begriffe sind Defaultwert und Initialwert. Diese werden für die weitere Verwendung wie folgt definiert:

Ein **Defaultwert** ist der Wert eines Modell-Parameters, der vom Parameterbesitzer, üblicherweise also einem SystemC-Modul, beim Erzeugen des Parameters gesetzt wird. Dies geschieht häufig über einen Konstruktor-Parameter. Der Defaultwert besitzt die niedrigste Präzedenz aller Parameterwerte, wird also von allen anderen überschrieben.

Ein **Initialwert** ist ein Parameterwert – üblicherweise von einer übergeordneten Instanz, also einem Parent-Modul oder einem Tool – der schon gesetzt werden kann, bevor der Modell-Parameter von seinem Besitzer erzeugt wird oder dessen Wert abgefragt wird. Ein Initialwert hat Vorrang vor einem potentiellen Defaultwert, obwohl der Defaultwert im zeitlichen Ablauf nach dem Initialwert angewendet werden kann. Wird versucht, einen Initialwert zu setzen, nachdem der Modell-Parameter bereits erzeugt wurde und potentiell einen Defaultwert besitzt, wird der Initialwert als normaler manipulativer Wertzugriff durchgeführt und überschreibt auch in dieser Situation den aktuellen Parameterwert und damit auch den Defaultwert.

Abschnitt 3.1.3 beinhaltet eine weitergehende Diskussion über verschiedene Rangfolgen dieser Werte zueinander.

3.1.2. Konfigurationsansätze

Als wichtige Erkenntnis stellt sich heraus, dass es zwei generell unterschiedliche Ansätze für die Konfiguration eines SystemC-Modells gibt (vgl. F1 in Tabelle 3.3 auf Seite 36): Implementierung eines Konfigurations-Interfaces und Class-Wrapper um Modellvariablen herum.

- Ein Modell, das mit einem **Konfigurations-Interface** ausgestattet ist, muss ein spezielles Interface mit einer oder mehreren Zugriffsfunktionen *implementieren oder verwenden*. Diese Funktionen sind beispielsweise vorgesehen, um konfigurierbare Eigenschaften des Modells zu deklarieren, zu setzen oder zu lesen. Den Zugriff auf den Wert regelt das Modell in diesen implementierten Interface-Funktionen potentiell selbst, wenn keine Implementierung vorgegeben ist. Ein einfaches Beispiel ist in Listing 3.4 gezeigt.

```
void set_param(string name, int value);
int  get_param(string name);
```

Listing 3.4: Beispiel für den Konfigurations-Interface-Ansatz

- Der zweite Ansatz sind **Class-Wrapper**, die auf einen oder mehrere Datentypen angewendet werden können. Ein Modell wird konfigurierbar durch *Instanziieren* eines Class-Wrapper-Objekts, das typischerweise eine Variable im Modell transparent ersetzt. Der Konfigurationsmechanismus bekommt automatisch Zugriff auf dieses instanziierte Objekt. Listing 3.5 zeigt ein Anwendungsbeispiel.

Grundsätzlich sind beide Ansätze in der Lage, vielfältige Merkmale zu unterstützen: Es können verschiedene Datentypen angeboten werden, C++- sowie SystemC- und benutzerdefinierte Datentypen sind möglich. Modell-Konfiguration und -Untersuchung (Lesen und Setzen

```
class A {
  param<int> my_param;
  param<int> my_reg;
};
```

Listing 3.5: Beispiel für den Class-Wrapper-Ansatz

von Werten) sind sowohl vor als auch während der Simulationsphase realisierbar. Sowohl externe Werkzeuge als auch andere Modelle können auf die Modell-Parameter zugreifen. Werte können im Modell-Code, in Konfigurationsdateien oder Datenbanken der Werkzeuge vordefiniert werden, und Simulationsstatus können exportiert und permanent gespeichert werden (vgl. [Mont10]). Eine Liste aller im System vorhandenen Parameter kann im Werkzeug oder in der Testbench angezeigt werden.

Die Verwendung eines Konfigurations-Interfaces ist für die Entwickler des Konfigurationsmechanismus mit wenig Aufwand verbunden, wenn sie keine Implementierung vorgeben. Es müssen lediglich das Interface und einige Regeln definiert werden, wogegen der Implementierungsaufwand für die geforderten Merkmale vom Modellentwickler (Benutzer) zu leisten ist. Dieser Ansatz ist fehleranfällig und für den Benutzer aufwändig. Des Weiteren darf der Mechanismus keine sicheren Annahmen über die tatsächlich implementierten Funktionen machen und muss somit die Merkmale einschränken. Würden die Merkmale nicht eingeschränkt werden, würde dem Benutzer viel Verantwortung für die Korrektheit seiner Implementierung aufgebürdet, und die Fehleranfälligkeit würde weiter erhöht werden.

Der Class-Wrapper-Ansatz ist für den Entwickler des Mechanismus wesentlich aufwändiger zu realisieren, da neben der Definition des obligatorischen Interfaces auch die Implementierung der Class-Wrapper bereitgestellt wird. Das bedeutet einen höheren Implementierungsaufwand auf Seite des Mechanismus, aber eine Vereinfachung auf Benutzerseite, zumindest bei mächtigeren Merkmalen des Konfigurationsmechanismus. Ein weiterer Vorteil ist, dass der Mechanismus die Kontrolle über die Implementierung hat und sich somit auf die Einhaltung der Regeln verlassen kann.

Der Stand der Technik hat gezeigt, dass die vielfältigen Möglichkeiten bei Verwendung des Konfigurations-Interface-Ansatzes nicht ausgeschöpft werden.

Als entscheidender Unterschied zwischen den beiden beschriebenen Ansätzen kann sich der Konfigurationsmechanismus beim Class-Wrapper-Ansatz auf das Verhalten verlassen: Das Benachrichtigen von Beobachtern von Modell-Parameter-Wertänderungen kann mit dem Konfigurations-Interface-Ansatz nicht automatisiert (und damit garantiert) werden. Der Grund hierfür ist, dass die Werte der Parameter nicht vom Konfigurationsmechanismus gekapselt werden, sondern vom Modell selbst verwaltet werden. Es obliegt dem Modell, bei jedem internen und externen Zugriff über das Konfigurations-Interface aktiv das entsprechende Benachrichtigungsereignis auszulösen. Falls dies nicht geschieht, verpassen Beobachter das Ereignis. Beim Class-Wrapper-Ansatz existiert das beschriebene Problem nicht, da

der Wert des Parameters durch den Mechanismus gekapselt ist. Jeder Zugriff, egal ob von extern über den Mechanismus oder von innerhalb des Modells direkt auf das Objekt, kann vom Mechanismus abgefangen und das Benachrichtigungsereignis automatisch ausgelöst werden.

Fazit

Der Class-Wrapper-Ansatz ist mächtiger als der Konfigurations-Interface-Ansatz. Da der hier vorgestellte Modell-Konfigurations- und Modell-Untersuchungs-Mechanismus Adapter zu möglichst vielen anderen Mechanismen bereitstellen soll, wird er intern sowie als native Standard-API den Class-Wrapper-Ansatz verwenden. Die Zuverlässigkeit des Class-Wrapper-Ansatzes wird für die Bereitstellung von Adaptern zu anderen Mechanismen unbedingt benötigt. Abschnitt 3.3.3 behandelt, wie es möglich ist, den Konfigurations-Interface-Ansatz in den Class-Wrapper-Ansatz umzusetzen.

3.1.3. Konfigurationsrangfolgen

Die Reviews der existierenden Konfigurationsmechanismen haben gezeigt, dass es zwei grundlegend verschiedene Rangfolgen gibt. Diese entstehen durch unterschiedliche Präzedenzen der manipulativen Wertzugriffe auf Modell-Parameter. Dieser Abschnitt zeigt diese Variationen der Rangfolgen und definiert einheitliche Begriffe.

Es gibt verschiedene **Typen manipulativer Wertzugriffe**:

- Das Setzen eines Initialwertes (vgl. Abschnitt 3.1.1),
- das Setzen eines Defaultwertes (vgl. Abschnitt 3.1.1) und
- eine sonstige Wertänderung (nach der Initialisierung).

Als Grundvoraussetzung für die folgenden Begriffsdefinitionen haben diese drei verschiedenen Wertzuweisungen unterschiedliche *Präzedenzen*, die (per Begriffsdefinition) in allen Mechanismen identisch sind: Initialwerte überschreiben Defaultwerte (unabhängig vom Zeitpunkt der Zuweisung) und sonstige Wertänderungen überschreiben die beiden anderen Werte, unter der Voraussetzung, dass dieses Merkmal vom Mechanismus unterstützt wird und der Wert nicht beispielsweise gesperrt (gelockt) ist.

Die Rangfolge der Zugriffe innerhalb eines Typs kann dagegen variieren:

Zeitliche Konfigurationsrangfolge

Manipulative Wertzugriffe haben eine **zeitliche Konfigurationsrangfolge**, wenn bei zeitlich nacheinander ausgeführten Wertzugriffen[3] des gleichen Typs die zeitliche Position des Zugriffs[4] den tatsächlichen Wert bestimmt. Der Tabelle 3.3 ist zu entnehmen, dass der überwiegende Teil der Konfigurationsmechanismen diese zeitliche Rangfolge für Wertzuweisungen

[3] Alle Zugriffe werden nacheinander ausgeführt, echte Parallelität wird in SystemC nicht unterstützt.
[4] Unter der zeitlichen Position eines Zugriffs könnte der früheste oder späteste Zugriff verstanden werden.

verwendet. Bei allen diesen Mechanismen bestimmt stets der zuletzt angewendete Zugriff den Wert.

Hierarchische Konfigurationsrangfolge

Manipulative Wertzugriffe haben eine **hierarchische Konfigurationsrangfolge**, wenn innerhalb des gleichen Typs von Wertzugriffen nicht der Zeitpunkt, sondern die hierarchische Position der setzenden Instanz den Ausschlag gibt, welchen Wert der Parameter annimmt. Die hierarchische Position eines Moduls ist hier die Position in der strukturellen SystemC-Modulhierarchie (vgl. [IEEE06]). Beispielsweise kann ein Modul einen Initialwert eines Parameters setzen, der sich in dessen Subsystem befindet. Dieser Initialwert überschreibt jeden innerhalb des Subsystems gesetzten Initialwert dieses Parameters, unabhängig von den Zeitpunkten der Schreibzugriffe[5]. Der im Abschnitt 2.6.6 beschriebene Konfigurationsmechanismus OVM verwendet diese Rangfolge.

Vergleich

Ein wesentlich unterschiedliches Verhalten zwischen den beiden Varianten der Rangfolge entsteht, wenn Modell-Parameter während der Konstruktion eines Subsystems dort sofort verwendet werden, das heißt die Initialwerte sofort im Konstruktor von Relevanz sind und sich beispielsweise auf die zu konstruierende Struktur des Systems auswirken. In dem Fall unterscheiden sich die Verfahren wie folgt: bei der zeitlichen Rangfolge wird typischerweise der zuletzt angewendete Initialwert als tatsächlicher Parameterwert zugewiesen. Dieser zuletzt angewendete Wert ist immer der des hierarchisch tiefsten Moduls, da die Konfiguration (das Setzen des Initialwertes) nicht zwischen dem Konstruieren des Child-Moduls (bzw. Subsystems) und dessen Verwenden des Modell-Parameters stattfinden kann (vgl. Abbildung 2.6 auf Seite 28). Deswegen kann die Konfiguration spätestens unmittelbar vor dem Konstruieren des Child-Moduls geschehen. Diese Konfiguration kann allerdings durch jeden unterhalb der aktuellen Hierarchiestufe gesetzten Initialwert überschrieben werden. Folglich ist eine sinnvolle Regel in Mechanismen, die die zeitliche Rangfolge verwenden, das Setzen von Initialwerten nur auf höchstem Hierarchielevel vorzunehmen.

Beim Verwenden von hierarchischer Rangfolge wird dieses Problem unterbunden. Dadurch gewinnt der Benutzer Komfort beim Erstellen und Konfigurieren von Modulen. Allerdings muss dem Mechanismus – den naheliegenden Wunsch vorausgesetzt, dass der Mechanismus die Hierarchie der setzenden Module und die hierarchische Position des Modell-Parameters automatisch erkennen soll – ebendiese hierarchische Position automatisch bekannt sein, was die Flexibilität des Mechanismus einschränkt. Bei OVM beispielsweise wird dieses Merkmal ermöglicht, indem das Benutzermodul von einem speziellen OVM-Modul (statt von dem

[5]Zur Verdeutlichung: Die Konfigurationsrangfolge hat keinen Einfluss auf die Präzedenz der verschiedenen Wertzugriffs-Typen (vgl. Grundvoraussetzung). Das heißt auch ein von einem hierarchisch höheren Modul gesetzter Initialwert wird von jeder sonstigen Wertänderung (nach der Initialisierung) überschrieben, auch wenn der von einer hierarchisch niedrigeren Position aus gesetzt wird.

Standard-SystemC-Modul) ableiten muss und der Wert des Modell-Parameters bei jedem lesenden Zugriff zeitaufwändig in der Hierarchie gesucht wird.

Fazit

Da die zeitliche Konfigurationsrangfolge weiter verbreitet ist, wird sie in dem hier vorgestellten Konfigurationsmechanismus verwendet und zwar in der naheliegenden Variante, dass der zuletzt gesetzte Wert relevant ist. Um das Interoperabilitätsziel dieser Arbeit für die unterschiedlichen Rangfolgevarianten sicherzustellen, werden im Abschnitt 3.3.4 verschiedene Ansätze diskutiert, wie auch die hierarchische Konfigurationsrangfolge unterstützt werden kann. Dort werden auch Regeln vorgestellt, die es erlauben, den Komfort und die Vorteile der hierarchischen Rangfolge mit minimalem Zusatzaufwand auch mit der zeitlichen Rangfolge nutzen zu können.

3.2. Anforderungen

Dieser Abschnitt beschreibt die Anforderungen, die an den in dieser Arbeit entwickelten universellen Konfigurationsmechanismus für die Modell-Konfiguration gestellt werden. Sie dienen dem Ziel dieser Arbeit, die Austauschbarkeit von Modellen zwischen verschiedenen Entwicklungsumgebungen zu verbessern.

Einige der folgenden Anforderungen tragen Merkmalmarkierungen in Klammern (F1 bis F14), die denen aus Tabelle 3.3 entsprechen und anzeigen, dass das entsprechende Merkmal mit der Anforderung abgedeckt ist. Wenn alle Anforderungen von dem universellen Konfigurationsmechanismus dieser Arbeit erfüllt werden, ist die Integration aller untersuchten Mechanismen möglich (der Nachweis wird in Abschnitt 5.2 geführt).

Abstrakte Anforderungen

Zunächst wird eine Liste meiner abstrakten Anforderungen (AA) präsentiert, die sich später in konkrete und detailliertere Anforderungen aufgliedern:

- AA1: *Aus dem Stand der Technik abgeleitete Merkmale*
  Die grundlegenden Anforderungen ergeben sich aus dem Stand der Technik in Abschnitt 2.6. Die Merkmale der dort untersuchten Mechanismen sollen unterstützt werden, folglich sind die dort erkannten Merkmale auch im universellen Mechanismus notwendig. Details werden bei den konkreten Anforderungen gelistet.

- AA2: *Verschiedene Konfigurationsansätze integrierbar*
  Eine besondere Kernanforderung ist die Notwendigkeit, verschiedene Konfigurationsansätze integrieren zu können (vgl. Abschnitt 3.1.2). Der universelle Konfigurationsmechanismus muss Adapter für verschiedene Modelle bereitstellen können, die entweder

den Ansatz Konfigurations-Interfaces oder den Class-Wrapper-Ansatz verwenden. Daraus folgt, dass die Integration verschiedener Ansätze in einer Simulation möglich ist. (F1)

- AA3: *Verschiedene Konfigurationsrangfolgen integrierbar*
  Der universelle Konfigurationsmechanismus muss zeitliche und hierarchische Konfigurationsrangfolgen unterstützen. Die Integration von Modellen – das können auch Subsysteme sein – soll unabhängig von der Konfigurationsrangfolge gemischt innerhalb einer Simulation sinnvoll möglich sein. (F14)

- AA4: *Erweiterbarkeit für nicht bedachte Merkmale*
  Ein erarbeiteter Stand der Technik kann unmöglich den Anspruch auf Vollständigkeit erheben, folglich werden potentiell nicht alle möglichen Merkmale erkannt, die entweder bereits existieren oder in zukünftigen Konfigurationsmechanismen anzutreffen sein werden. Deswegen muss der universelle Mechanismus erweiterbar sein. Eine Erweiterung soll existierende Lösungen so wenig wie möglich beeinflussen. Es wäre von Vorteil, vorhandene Modelle bei einer Änderung des unterliegenden Mechanismus nicht neu kompilieren zu müssen.

- AA5: *Flexible interne API für Adapter*
  Wenn ein neuer Adapter für einen bis dahin nicht unterstützten proprietären Konfigurationsmechanismus in den universellen Mechanismus eingebunden wird, soll dies ohne große interne Änderungen am universellen Mechanismus möglich sein. Die interne API zum Anschluss der Adapter muss also hochflexibel sein und möglichst kein Neukompilieren von vorhandenen Modellen notwendig machen.

- AA6: *Modifikation von Modell-Quellcode nicht notwendig oder gering*
  Um das Hauptziel dieser Arbeit unverfälscht zu erreichen, sollte ein Modell, das einen proprietären Konfigurationsmechanismus nutzt bzw. für diesen entwickelt wurde, bei der Integration in den universellen Mechanismus nicht verändert werden müssen. Für ausschließlich binär vorliegende Modelle dürfte diese Anforderung kaum zu erfüllen sein, im Quellcode vorliegende Modelle sollten jedoch ohne Codeänderungen mit einem Adapter zum universellen Mechanismus neu kompiliert werden können. Nur in Ausnahmefällen, wenn eine Integration nicht anders möglich ist, sind Codeänderungen akzeptabel.

- AA7: *Proprietäres Modell in universeller Entwicklungsumgebung (PIU)*
  Ein Modell, das einen proprietären Mechanismus verwendet, muss in eine den universellen Mechanismus verwendende Simulation einzubinden sein (vgl. Abschnitt 3.3.1). Der umgekehrte Weg (Universelles Modell in proprietärer Entwicklungsumgebung (UIP), vgl. Abschnitt 3.3.1) ist wünschenswert, aber keine Anforderung.

Konkrete Anforderungen

Es folgt eine Liste von konkreten Anforderungen (KA), die abstrakte Anforderungen konkretisieren oder weitere detaillierte Anforderungen aus dem Stand der Technik hinzufügen. Jede Anforderung muss vom universellen Mechanismus entweder direkt unterstützt werden oder mit einer Erweiterung realisierbar sein. Wenn nicht anders angegeben, sollen alle Merkmale unabhängig von der Aufgabe und Position des jeweiligen Aufrufers zur Verfügung stehen, das heißt sie stehen sowohl dem Modell selbst als auch dem Tool zur Verfügung. Die Anforderungen KA1 bis KA15 sind eigene Anforderungen, KA16 bis KA21 sind von den Diskussionen in der CCI und ihrem Anforderungsdokument [OSCI09]^i abgeleitet.

- KA1: *Verfügbarkeit*
  Der Konfigurationsmechanismus samt den Modell-Parametern ist mindestens während der Ausführung der SystemC-Anwendung verfügbar. Das bedeutet insbesondere die Verfügbarkeit in der `sc_main`-Funktion während der Elaborationsphase und während der Simulationsphase (Laufzeitkonfiguration). Darüber hinaus wäre die Verfügbarkeit bereits zur statischen Initialisierung, also während der gesamten Programmphase, hilfreich.

- KA2: *Modell-Parameternamen*
  Modell-Parameter werden über hierarchische Namen identifiziert. Es sollen auch Top-Level-Namen ohne Hierarchie möglich sein.

- KA3: *Datentypen*
  Modell-Parameter unterstützen alle primitiven C++-, SystemC- sowie benutzerdefinierte Datentypen. Alle nicht direkt unterstützten Datentypen können (solange sie kopierbar sind) als benutzerdefinierte Typen nachgerüstet werden (F2, F3).

- KA4: *Portable Wertrepräsentation*
  Der Wert eines Parameters kann portabel[6] dargestellt werden und ist zu den entsprechenden Parametertypen in beide Richtungen konvertierbar.

- KA5: *Dateiausgabe*
  Das Exportieren von Konfigurationswerten in Dateien unterschiedlichen Typs ist während der gesamten Programmphase möglich[7] (F4).

- KA6: *Dateieingabe*
  Das Einlesen von Konfigurationsdateien und das Setzen der dort enthaltenen Werte ist während der gesamten Programmphase möglich[7], was vor allem vor und während der Elaborationsphase sinnvoll ist.

[6] Die portable Repräsentation eines Parameterwertes bedeutet, dass der Wert sprachen- und plattformunabhängig (z.B. als Zeichenkette) dargestellt werden kann.

[7] Diese Anforderung kann mit Hilfe von anderen Kernmerkmale als Erweiterung (Bequemlichkeitsfunktion) hinzugefügt werden.

- KA7: *Globales Verzeichnis*
  Ein Verzeichnis sämtlicher (nicht versteckter) Modell-Parameter ist während der gesamten Programmphase global verfügbar, und ihre Eigenschaften (Werte etc.) können abgefragt werden (F5). In diesem Verzeichnis kann nach Parametern gesucht werden.

- KA8: *Wahlfreier Zugriff*
  Auf jeden (nicht versteckten) Modell-Parameter kann global zugegriffen werden, wenn der Parametername bekannt ist. Namen können beispielsweise über das globale Verzeichnis abgefragt werden.

- KA9: Manipulativer Wertzugriff, *Defaultwert*
  Beim Erzeugen eines Modell-Parameters kann ein Defaultwert angegeben werden. Die Präzedenzen aus Abschnitt 3.1.3 werden eingehalten (F6).

- KA10: Manipulativer Wertzugriff, *Initialwert*
  Initialwerte können während der gesamten Verfügbarkeit des Mechanismus gesetzt werden. Die Präzedenzen aus Abschnitt 3.1.3 werden eingehalten (F7). Initialwerte bleiben erhalten, auch wenn der Parameter gelöscht wird.

- KA11: *Wertneutraler Zugriff*
  Wertneutrale Zugriffe auf Modell-Parameter sind während der Existenz des jeweiligen Parameters möglich (F8). Dieser Zugriff sollte möglichst effizient sein.

- KA12: Manipulativer Wertzugriff, *sonstige Wertänderung*
  Manipulative Wertzugriffe, insbesondere Wertänderungen, sind während der Existenz des jeweiligen Parameters möglich (F9, F10). Dieser Zugriff sollte möglichst effizient sein.

- KA13: *Benachrichtigungen*
  Ein Beobachter eines Modell-Parameters kann sich beim Mechanismus registrieren, sodass er unmittelbar (z.B. ohne Delta-Schritt- oder Zeitverzögerung) über jede Wertänderung mit einem Callback benachrichtigt wird (post_write). Weitere verschiedene Benachrichtigungsarten zu verschiedenen Ereignissen sind von Vorteil, z.B. vor einem manipulativen Wertzugriff (pre_write) und bei einem wertneutralen Zugriff (pre_read) (F11, F12). Manipulative Wertzugriffe können vom Beobachter abgelehnt werden.

- KA14: *Gruppen-Zugriff*
  Auf Gruppen von Modell-Parametern kann mit Wildcards[8] oder mit regulären Ausdrücken[9] (in Tabelle 3.3 RegExp abgekürzt) zugegriffen werden (F13).

[8] Wildcards (Joker) sind Platzhalter (z.B. Sternchen (*)) für andere Zeichen oder beliebig viele andere Zeichen.

[9] Reguläre Ausdrücke sind nach bestimmten Regeln interpretierte Zeichenketten, die eine Menge anderer Zeichenketten beschreiben und einen mächtigeren Aufbau als Wildcards erlauben.

- KA15: *Unauffälligkeit in SystemC*
  Der gesamte Mechanismus soll sich unauffällig in SystemC eingliedern. Das bedeutet die SystemC-spezifischen Eigenheiten sollen umfassend unterstützt werden, und trotzdem soll der Mechanismus für eine SystemC-Analyse externer Tools möglichst unsichtbar sein. Beispielsweise soll die Verwendung von SystemC-Modulen (`sc_module`) vermieden werden.

- KA16: *Sperren von Modell-Parametern gegen manipulative Wertzugriffe*
  Modell-Parameter können zu beliebigen Zeitpunkten[10] gegen manipulative Wertzugriffe gesperrt werden[11].

- KA17: *Sperren von Initialwerten*
  Ein einmal gesetzter Initialwert kann gegen Manipulation gesperrt werden, sodass er auf jeden Fall angewendet wird, wenn der Parameter erzeugt wird[11].

- KA18: *Versteckte Modell-Parameter*
  Modell-Parameter können versteckt oder privat erzeugt werden, sodass der Zugriff von außerhalb des besitzenden Modells eingeschränkt ist[11]. Beispielsweise werden diese Parameter nicht im globalen Verzeichnis geführt.

- KA19: *Nicht verwendete Initialwerte*
  Nicht verwendete Initialwerte können identifiziert werden. Damit können beispielsweise nach der Simulation oder Elaboration Fehler beim Setzen von Initialwerten erkannt werden.

- KA20: *Zusatzinformationen*
  Neben einem Namen und einem Wert können Parameter weitere Informationen enthalten, beispielsweise eine Dokumentation.

- KA21: *Wert-Herkunft*
  Von einem Parameter kann die Herkunft des Wertes abgefragt werden, d.h. welches Modul oder welcher Prozess den Wert zuletzt geschrieben oder gelesen hat.

3.3. Konfigurations-Interoperabilität

Wie bereits zuvor erläutert, soll der in dieser Arbeit vorgestellte Konfigurationsmechanismus einen Teilaspekt der in Abschnitt 2.4 eingeführten Meta-Interoperabilität[12] ermöglichen, die

[10] Inklusive direkt bei oder nach der Konstruktion

[11] Dies soll kein Schutz gegen bösartige Tools oder Modelle sein, die die vorgesehenen Mechanismen und APIs umgehen, sondern die Regeln beachtende Tools und Modelle davon abhalten, Parameter entgegen ihrer Bestimmung zu verwenden.

[12] Neben der Konfigurations-Interoperabilität sind weitere Meta-Interoperabilitätstypen für beispielsweise Analyse- und Debugverfahren denkbar.

Konfigurations-Interoperabilität: darunter wird die Zusammenarbeit zwischen Meta-Modellen und Tools bezüglich der in Abschnitt 2.3 eingeführten Modell-Konfiguration und Modell-Untersuchung verstanden.

Um die Problematik der Konfigurations-Interoperabilität zu verdeutlichen, werden zunächst relevante Situationen beschrieben und im folgenden Abschnitt verschiedene Integrationen daraus abgeleitet. Die im Allgemeinen aktuell vorherrschende Situation ist in Abbildung 3.6 schematisch abgebildet. Es handelt sich um ein **proprietäres Modell**, das speziell für eine **proprietäre Entwicklungsumgebung** (Beispiele siehe Stand der Technik in Abschnitt 2.6) geschrieben wird und die dort vorhandenen Mechanismen zur Konfiguration verwendet. Ein solches Modell ist im Allgemeinen bezüglich der Konfiguration nicht kompatibel zu Entwicklungsumgebungen anderer Hersteller.

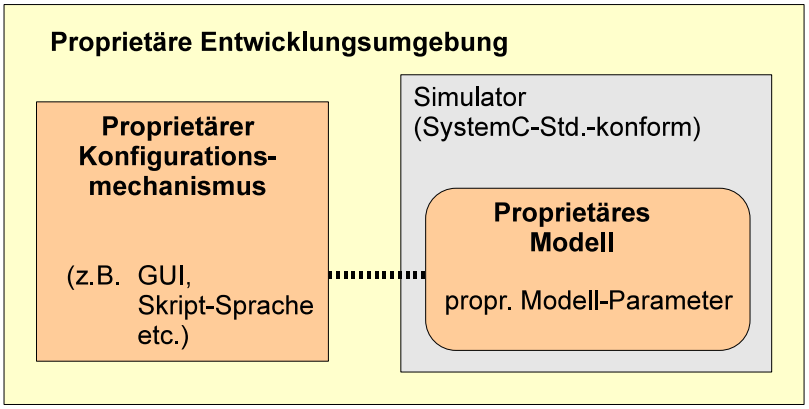


Abbildung 3.6.: Ausgangssituation Konfigurations-Interoperabilität: Proprietäre Systeme

Eines der in dieser Arbeit verfolgten Ziele ist ein herstellerunabhängiges **universelles Modell**, das in eine **universelle Entwicklungsumgebung** eingebunden und konfiguriert werden kann. Abbildung 3.7 zeigt den grundlegenden Aufbau. Zunächst sieht dieser Ansatz aus wie ein weiterer proprietärer Mechanismus. Allerdings soll die universelle Entwicklungsumgebung die Merkmale von anderen Konfigurationsmechanismen vereinen und sehr flexibel und erweiterbar sein und somit verschiedene Arten der im folgenden Abschnitt erläuterten Integrationen ermöglichen.

3.3.1. Integrationsverfahren

Für die angestrebte Konfigurations-Interoperabilität sind zwei grundlegend verschiedene Integrationsrichtungen denkbar: Das Hauptziel ist, proprietäre Modelle in einer universellen Entwicklungsumgebung zu verwenden (AA7). Mit einigen Einschränkungen kann es zusätzlich möglich sein, fremde (z.B. universelle oder andere proprietäre) Modelle in einer proprietären Entwicklungsumgebung zu verwenden.

PIU-Integration: Proprietäres Modell in universeller Entwicklungsumgebung (PIU)
Die als Hauptziel verfolgte Integrationsrichtung besteht aus der in Abbildung 3.8 dargestell-

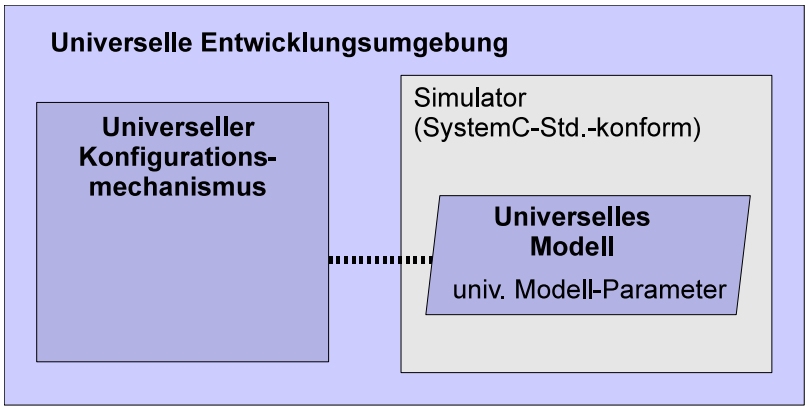


Abbildung 3.7.: Einfacher Aufbau einer universellen Entwicklungsumgebung mit einem universellen Modell

ten Verwendung von proprietären Modellen in einer (im Rahmen dieser Arbeit entwickelten) universellen Entwicklungsumgebung. Es ist ein wesentliches Ziel, dass der Quellcode des proprietären Modells für diese Integration nicht modifiziert werden muss (AA6). Für das Modell soll die universelle Umgebung die erwartete proprietäre transparent ersetzen. Das bedeutet, dass alle Eigenschaften des proprietären Konfigurationsmechanismus auch im universellen vorhanden sein sollten und über einen Adapter auf die gleiche Weise funktionieren sollten. Außerdem ist es wünschenswert, dass Modelle verschiedener Konfigurationsmechanismen (z.B. Modell A und B in Abbildung 3.8) in einer universellen Simulation vereinigt werden können.

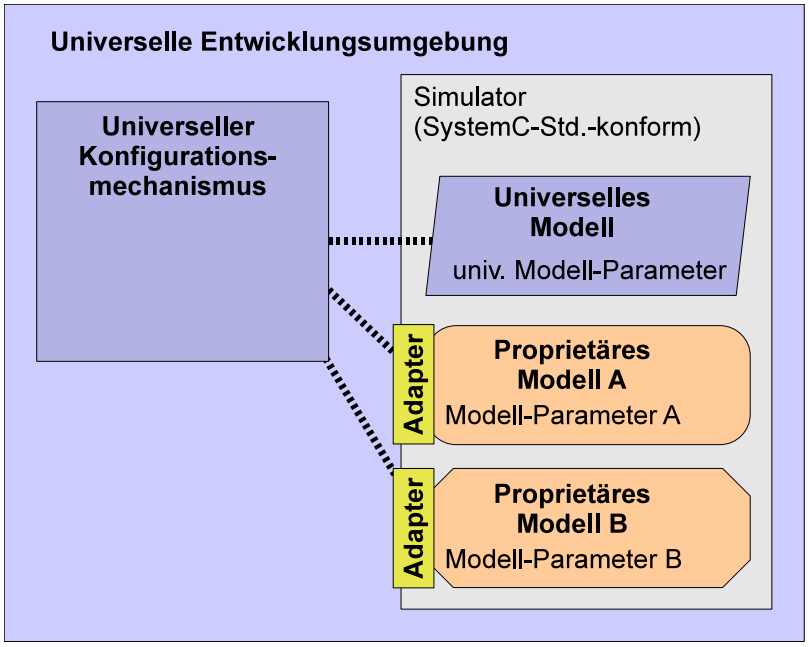


Abbildung 3.8.: PIU-Integration: Universelle Entwicklungsumgebung mit einem universellen und zwei verschiedenen integrierten proprietären Modellen

UIP-Integration: Universelles Modell in proprietärer Entwicklungsumgebung (UIP) Eine je nach den Merkmalen der verschiedenen Mechanismen ebenfalls denkbare Integrationsrichtung ist die folgende: Der Benutzer soll ein Modell mit universellen Modell-Parametern entwickeln können, welches dann in der proprietären Ziel-Entwicklungsumgebung wie ein proprietäres Modell eingebunden werden kann. Das bedeutet, dass das universelle Modell in der (z.B.) grafischen Oberfläche genauso angezeigt wird wie proprietäre Modelle. Der Zugriff soll in beiden Richtungen kompatibel sein; das universelle Modell soll auf proprietäre Konfigurationselemente zugreifen können – sofern dieser Zugriff im proprietären Mechanismus unterstützt wird – und umgekehrt. Diese Interoperabilität ist eingeschränkt durch die Merkmale des proprietären Konfigurationsmechanismus. Ein Merkmal, das vom Konfigurationsmechanismus der umgebenden Entwicklungsumgebung nicht zur Verfügung gestellt wird, vom universellen Modell aber erwartet wird, steht entweder nicht zur Verfügung oder muss vom Adapter, wenn möglich, transparent hinzugefügt werden. Abbildung 3.9 zeigt ein schematisches Beispiel mit einem nativen und einem universellen Modell in einer proprietären Entwicklungsumgebung. Die Möglichkeiten der technischen Umsetzung im Rahmen des entwickelten Konfigurationsmechanismus sind in Abschnitt 5.9 erläutert. Der Quellcode des universellen Modells sollte für die Integration möglichst nicht modifiziert werden.

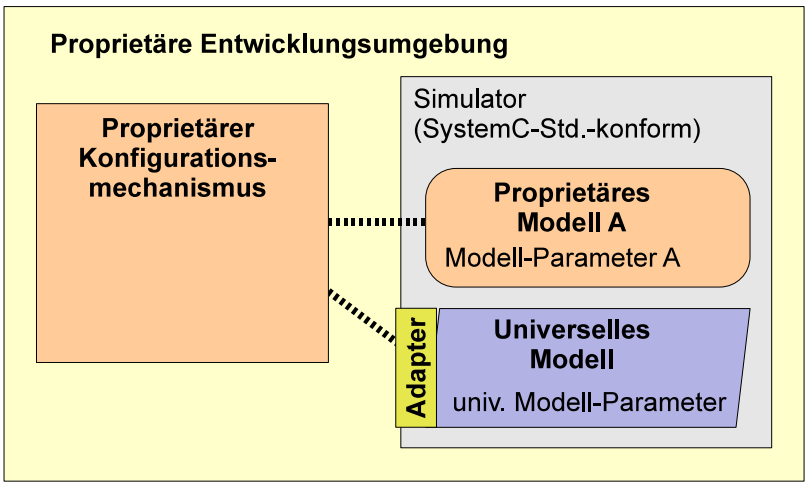


Abbildung 3.9.: UIP-Integration: Proprietäre Entwicklungsumgebung mit einem nativen und einem integrierten universellen Modell

PIU- + UIP-Integration Wenn für verschiedene Mechanismen die PIU- und UIP-Integrationen realisiert sind, ist der letzte Schritt die Kombination der verschiedenen Integrationsrichtungen unterschiedlicher Mechanismen[13]. Das ermöglicht dann die Konfiguration beliebiger Modelle in einer (mit der UIP-Integration unterstützten) Umgebung eines beliebigen Herstellers. Abbildung 3.10 zeigt einen Beispielaufbau: Der Adapter von Modell B übersetzt die Konfiguration in den universellen Mechanismus, der hier als Middleware fun-

[13]Das große Integrationsbeispiel in Abschnitt 6.6 demonstriert einen solchen Anwendungsfall.

giert und mit einem weiteren Adapter zum proprietären Mechanismus A übersetzt. Auch die Möglichkeiten dieses Verfahrens werden beschränkt durch die Merkmale des proprietären Konfigurationsmechanismus der umgebenden Entwicklungsumgebung. Zusätzliche Merkmale, die nicht ohne Unterstützung der Entwicklungsumgebung in der Middleware bereitgestellt werden können, stehen den Modellen nicht zur Verfügung und bedürfen einer möglichst toleranten Fehlerbehandlung in den Adaptern.

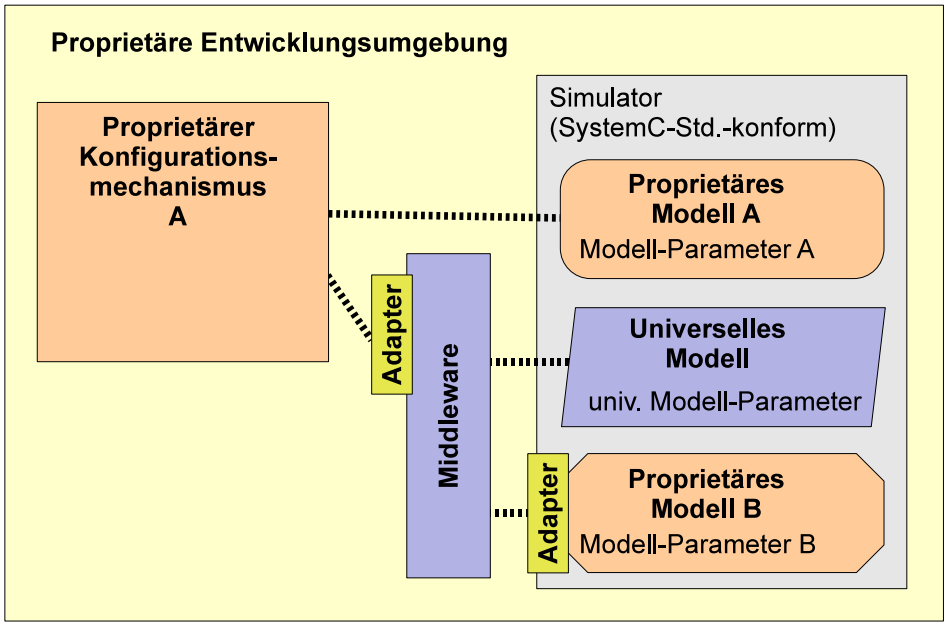


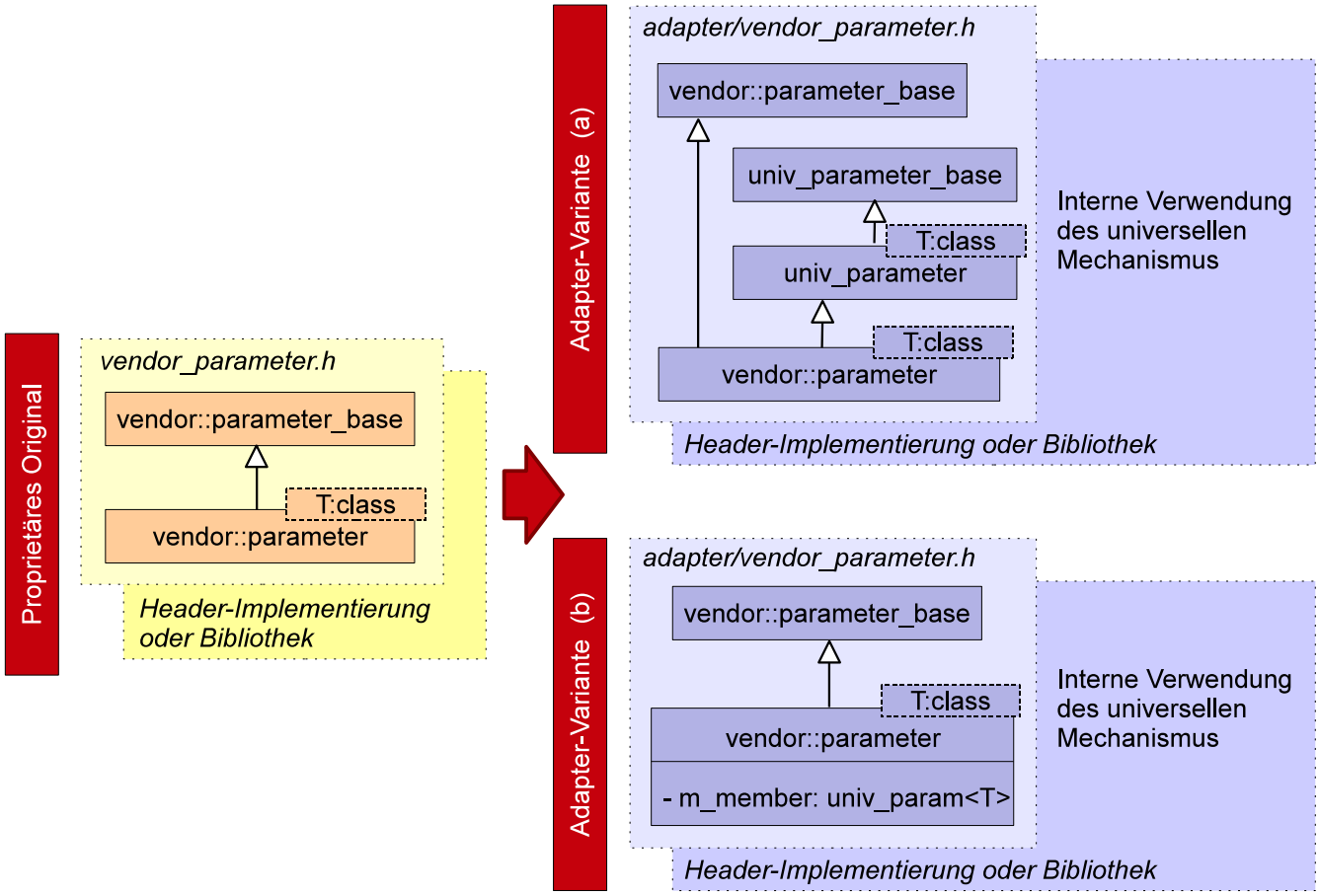
Abbildung 3.10.: PIU- + UIP-Integration: Proprietäre Entwicklungsumgebung mit einem universellen und zwei verschiedenen integrierten proprietären Modellen

Die folgenden Abschnitte beschreiben einige grundlegende Verfahren, mit denen die verschiedenen im Abschnitt 3.1 analysierten Mechanismen in einen universellen Mechanismus integriert werden können.

3.3.2. Class-Wrapper integrieren

Im Abschnitt 3.1.2 wurde bereits erläutert, dass der hier vorgestellte universelle Konfigurationsmechanismus intern Class-Wrapper verwendet. Dieser Abschnitt zeigt grob technische Möglichkeiten in C++/SystemC, mit denen ein fremder Class-Wrapper-Konfigurationsmechanismus mit der PIU-Integration in den universellen (Class-Wrapper-)Konfigurationsmechanismus integriert bzw. die umgekehrte UIP-Integration realisiert werden kann. Im Kapitel 6 werden einige konkrete Realisierungen detaillierter vorgestellt. Konkrete Realisierungen der Integration können im Detail von den hier vorgestellten idealisierten Mustern abweichen, wenn zusätzlich Hierarchien, Namensräume oder Ähnliches die Integration erschweren.

PIU-Integration Abbildung 3.11 zeigt Möglichkeiten, mit denen proprietäre Parameter in einem proprietären Modell ersetzt werden können, indem lediglich Compiler- oder Link-



Compiler/Linker:
Include-/Linker-Verzeichnis:
Parameter-Hersteller-Verzeichnis

Benutzer-Code:
```
#include "vendor_parameter.h"
struct my_module {
  vendor::parameter<int> myparam;
};
```

Compiler/Linker:
Include-/Linker-Verzeichnis:
Adapter-Verzeichnis

Benutzer-Code:
```
#include "vendor_parameter.h"
struct my_module {
  vendor::parameter<int> myparam;
};
```

Abbildung 3.11.: PIU-Integration Class-Wrapper.
Links: Originaler Klassenaufbau und Implementierung (oben) samt Compiler-Einstellungen und Benutzer-Code (unten) des proprietären Modells bzw. Parameters.
Rechts: Zwei Adapter-Varianten (a) und (b) mit modifiziertem Klassenaufbau (oben) und den Auswirkungen auf den Benutzer (unten).
Legende: Die Rechtecke, die Header-Dateien mit Klassendiagrammen darstellen, sind jeweils hinterlegt mit dunkleren Rechtecken, die deren Implementierung darstellen.

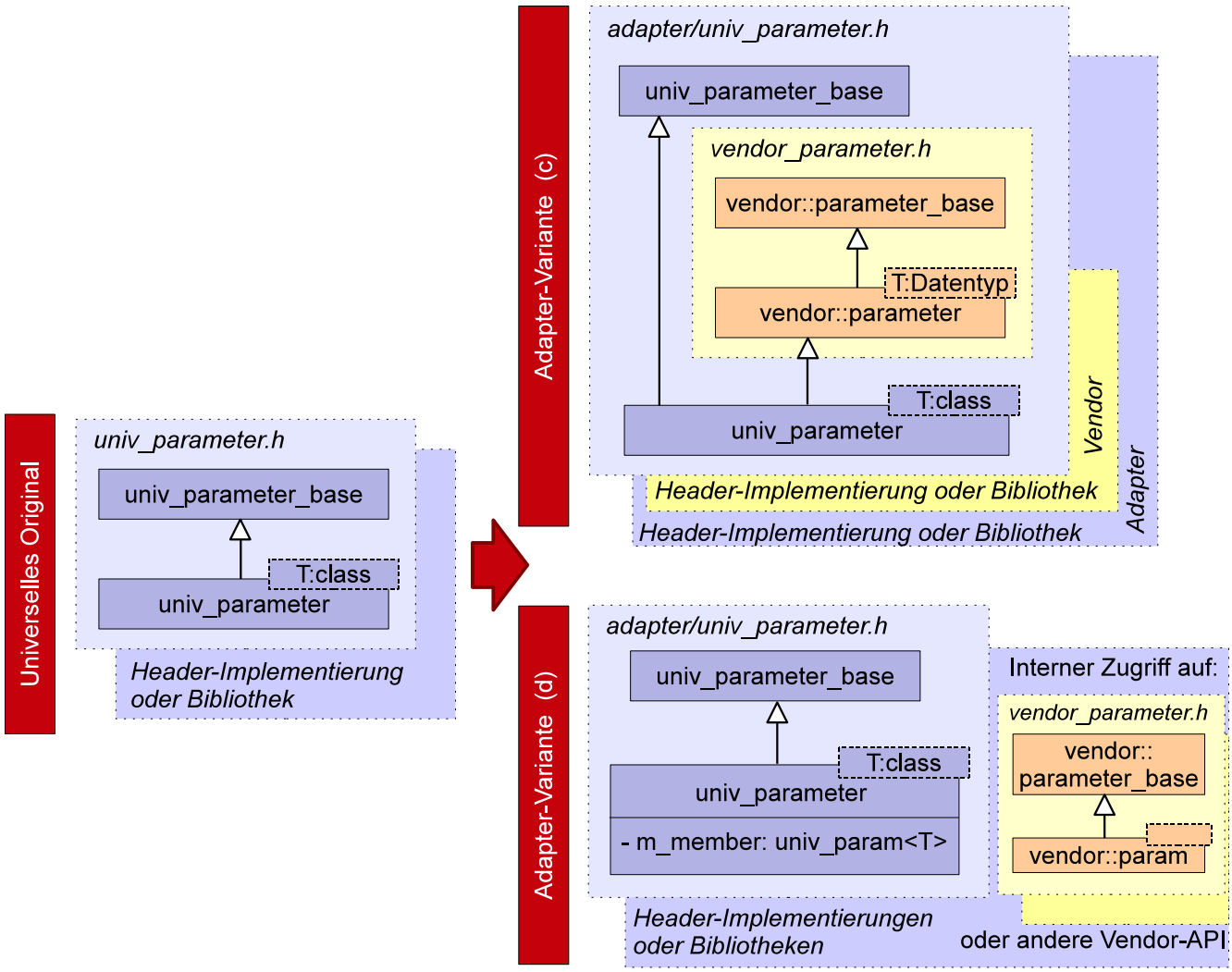
er-Einstellungen geändert werden. Adapter-Klassen[14] ersetzen die proprietären Parameter und verbinden das Modell zum universellen Konfigurationsmechanismus. Wenn die Adapter-Klasse in einer Datei mit gleichem Namen wie die Originalparameter zur Verfügung gestellt wird, kann der Benutzer-Code sogar unmodifiziert bleiben. Die Abbildung zeigt auf der linken Seite das Beispiel einer proprietären, mit einem Datentyp Template-parametrisierbaren Parameter-Klasse, die von einer nicht typabhängigen Basisklasse ableitet. Dieser Aufbau ist typisch für den Class-Wrapper-Ansatz[15]. Als Folge muss zumindest ein Teil der Implementierung im Header zur Verfügung stehen (Template-Implementierungen können nicht in Bibliotheken bereitgestellt werden), wodurch der Austausch der Parameter ein Neukompilieren des Benutzermodells bedingt. In der Adapter-Variante (a) leitet die Adapter-Klasse von dem universellen Parameter ab, wodurch sie dessen Eigenschaften erbt und beispielsweise lediglich zwischen Funktionsaufrufen übersetzen und Benachrichtigungen weitergeben muss. Alternativ ist Adapter-Variante (b) möglich, welche die Umsetzung zu dem universellen Parameter innerhalb der Implementierung der Klasse `vendor::parameter<T>` vornimmt und den universellen Parameter als Member speichert. Der untere Teil der Abbildung zeigt die – von der verwendeten Adapter-Variante unabhängigen – Auswirkungen auf den Benutzer: Es muss lediglich beim Kompilieren des Modells ein zusätzlicher bzw. anderer Include-Pfad sowie evtl. eine zusätzliche bzw. andere Bibliothek gelinkt werden. Der Modell-Quellcode bleibt unverändert; dieser einfache Vorgang ist folglich auch automatisierbar.

UIP-Integration Für eine UIP-Integration müssen universelle Parameter im universellen Modell mit einem Adapter in proprietäre Parameter übersetzt werden. Je nach Verfahren des proprietären Mechanismus kann das unterschiedlich anspruchsvoll funktionieren oder auch gar nicht möglich sein (vgl. Abschnitt 3.3.5). Um beispielsweise Parameter anzuzeigen, auszulesen oder mit Initialwerten zu belegen, kann eine Entwicklungsumgebung auf unterschiedliche Art und Weise an Informationen über die in den Modellen existierenden Parameter erlangen, zum Beispiel

- durch Quellcode-Parsing oder
- probeweises Ausführen des Modells oder des kompletten Systems (z.B. bis zur Elaborationsphase) für Informationen vor Laufzeit der Simulation,
- durch Abfragen der Parameter während der vom Benutzer durchgeführten Simulation (Programmphase) oder
- durch manuelles Registrieren von Parametern durch den Benutzer.

Der Konfigurationsmechanismus stellt unterschiedliche Anforderungen an die Integration: Beim naiven Quellcode-Parsing muss die Verwendung der Parameter im Quellcode identisch

[14] Die hier sog. Adapter-Klasse ist eine weitere Wrapper-Ebene um den (Class-Wrapper-)Parameter.
[15] Eine vom Typ unabhängige Basisklasse ist sinnvoll, damit Gemeinsamkeiten von Parametern verschiedenen Typs in C++ auf gleiche Art und Weise behandelt werden können.



Compiler/Linker:
Include-/Linker-Verzeichnis:
Universal-Parameter-Verzeichnis

Benutzer-Code:

```
#include "univ_parameter.h"
struct my_module {
  univ_parameter<int> myparam;
};
```

Compiler/Linker:
Include-/Linker-Verzeichnis:
Adapter-Verzeichnis

Benutzer-Code:

```
#include "univ_parameter.h"
struct my_module {
  univ_parameter<int> myparam;
};
```

Abbildung 3.12.: UIP-Integration Class-Wrapper.
Links: Originaler Klassenaufbau und Implementierung (oben) mit Compiler-Einstellungen und Benutzer-Code (unten) des universellen Modells bzw. Parameters.
Rechts: Zwei Adapter-Varianten (c) und (d) mit modifiziertem Klassenaufbau (oben) und den Auswirkungen auf den Benutzer (unten).
Legende: Die Rechtecke, die Header-Dateien mit Klassendiagrammen darstellen, sind jeweils hinterlegt mit dunkleren Rechtecken, die deren Implementierung darstellen.

aussehen (gleicher Klassenname der Parameter und u.U. gleicher Namensraum), komplexere Lösungen (komplexes Quellcode-Parsing, Kompilieren und evtl. probeweises Ausführen) können Klassenhierarchien erkennen, sodass auch von den Parametern abgeleitete Klassen als Parameter erkannt werden. Die Verwendung einer API während der Programmphase bietet die größte Flexibilität für die Integration, da einfach ein Interface implementiert oder verwendet werden kann.

Die UIP-Integration erfordert generell vielfältige Ideen, deswegen kann Abbildung 3.12 nur einen groben idealisierten Überblick über zwei generell mögliche Adapter-Varianten bieten: Auf der linken Seite ist der vom universellen Modell verwendete Class-Wrapper-Parameter sowie die Benutzereinstellungen und der Benutzer-Code gezeigt. Auf der rechten Seite sind die zwei Adapter-Varianten (c) und (d) gezeigt. Variante (c) ist ein universeller Parameter, der von der proprietären Parameterklasse ableitet. Diese Variante ist bei hinreichenden Gemeinsamkeiten sinnvoll und anwendbar, wenn der proprietäre Mechanismus die Klassenhierarchie erkennen kann. Variante (d) verpackt den adaptierten Parameter – wie schon bei der PIU-Integration – in einem Klassenmember. Intern kann dann die zur Verfügung stehende, hier nicht näher bezeichnete, proprietäre API verwendet werden. Da auch hier ein Teil der Implementierung im Header zur Verfügung stehen muss, ist ein Neukompilieren der Modelle notwendig, wenn der Konfigurationsmechanismus getauscht wird. Dadurch sind die Auswirkungen auf den Benutzer ähnlich den in der PIU-Integration gezeigten.

Alle Benachrichtigungen sind in beiden Integrationsverfahren transparent synchronisierbar, da die Kontrolle über den Wert des Modell-Parameters jeweils dem Adapter und dem universellen Parameter obliegt.

3.3.3. Konfigurations-Interface integrieren

Dieser Abschnitt stellt grob verschiedene technische Möglichkeiten in C++/SystemC vor, wie ein proprietärer Konfigurationsmechanismus mit Konfigurations-Interfaces in universelle Class-Wrapper integriert werden kann und umgekehrt. Auch hierfür werden im Kapitel 6 konkrete Realisierungen vorgestellt, die im Detail von den hier vorgestellten idealisierten Mustern abweichen können.

Um die verschiedenen Konfigurationsverfahren aufeinander abzubilden, werden die Benachrichtigungen bedeutsam. Für die verschiedenen Integrationen PIU und UIP sind beide Abbildungsrichtungen notwendig: Für eine PIU-Integration ist die Richtung Konfigurations-Interface nach Class-Wrapper umzusetzen, für die UIP-Integration die Richtung Class-Wrapper nach Konfigurations-Interface. Konkret sind hierbei besonders zwei Benachrichtigungstypen wichtig: Ein in dem einen Verfahren durchgeführter manipulativer Wertzugriff muss für die Synchronisation der Änderung an das andere Verfahren weitergegeben werden. Dafür wird eine Benachrichtigung nach dem Zugriff benötigt (post_write-Benachrichtigung). Bei einem lesenden wertneutralen Zugriff wird eine Benachrichtigung vor dem tatsächlichen Lesevorgang (pre_read-Benachrichtigung) benötigt, um potentiell unterschiedliche Status der beiden Parameter-Repräsentationen zu synchronisieren. Eine detaillierte Einführung und

Beschreibung der Realisierung dieser und weiterer Benachrichtigungstypen als Callbacks im universellen Konfigurationsmechanismus wird in Abschnitt 5.3 vorgenommen[16].

PIU-Integration Die Abbildungen 3.13 und 3.14 zeigen Adapter-Varianten, mit denen proprietäre Konfigurations-Interface-Parameter in das universelle (Class-Wrapper-)System abgebildet werden können, für die also eine Umsetzung in Class-Wrapper-Parameter notwendig ist. Die beiden Varianten unterscheiden sich im proprietären Quell-Mechanismus: Entweder ist die Implementierung des Konfigurations-Interfaces durch den Mechanismus vorgegeben (Abbildung 3.13) oder es ist nur das Interface gegeben, das vom Benutzer selbst implementiert werden muss (Abbildung 3.14).

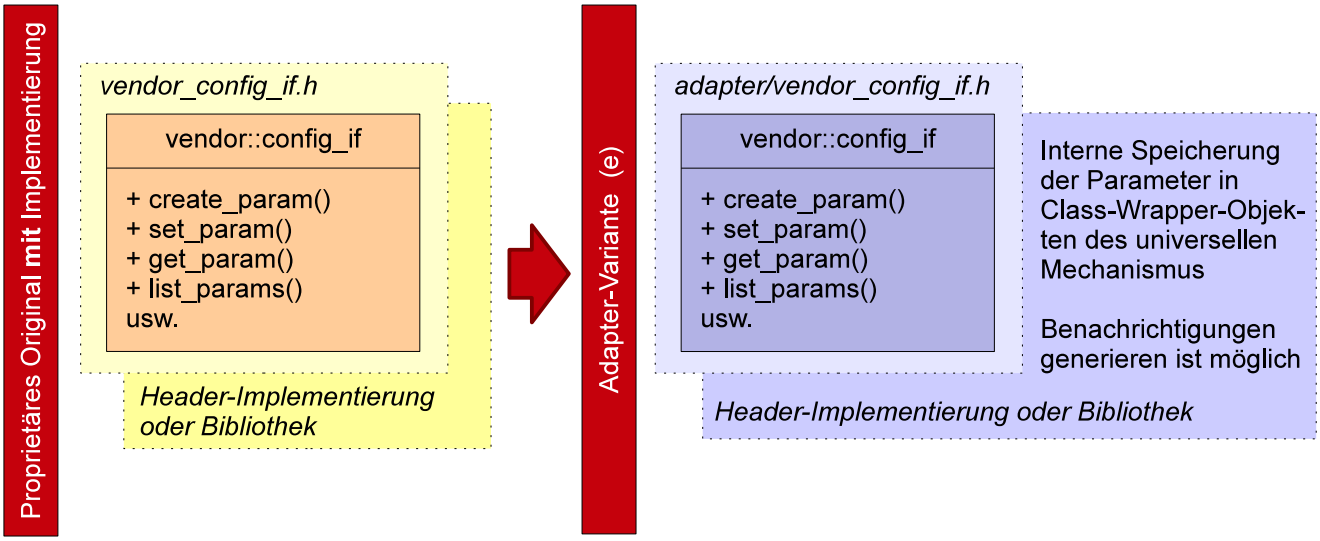
Die hier dargestellten Varianten unterscheiden nicht zwischen den beiden potentiell vom proprietären Mechanismus gewählten Optionen, ob entweder das Modell (SystemC-Modul) vom Konfigurations-Interface *ableitet* (und evtl. die Funktionen implementieren muss) oder ob es sich bei dem Interface um eine von dem Modell zunächst unabhängige Klasse handelt, die innerhalb des Modells *verwendet* werden muss. Obwohl die Konzepte zunächst unterschiedlich sind, sind die praktischen Auswirkungen auf den Adapter minimal. In den beiden Abbildungen sind diese verschiedenen Optionen jeweils im Benutzer-Code angedeutet: Abbildung 3.13 zeigt die Optionen als alternative Code-Auszüge, Abbildung 3.14 deutet den Unterschied mit einem Kommentar bei der Ableitung an.

Wenn der Konfigurations-Interface-Mechanismus die Implementierung anbietet, kann die Integration mit einem Adapter (e) nach Abbildung 3.13 realisiert werden. Die Interface-Klasse wird unverändert übernommen und deren Implementierung so verändert, dass die Speicherung der Modell-Parameter in Class-Wrapper-Parametern des universellen Mechanismus stattfindet. Durch einfache Änderungen beim Kompilieren (Änderung des Include- und Bibliotheks-Verzeichnisses) ist zwar ein Neukomilieren, aber keine Benutzer-Code-Änderung notwendig.

Die Generierung von Benachrichtigungen für beide Mechanismen ist – falls vom jeweiligen Mechanismus unterstützt bzw. gefordert – problemlos möglich, da die Kontrolle über die Werte und Zustände beider Parameter-Repräsentationen dem Adapter obliegt. Alle Zugriffe – lesende wie schreibende – auf das Interface können in der Implementierung der Interface-Funktion direkt an die Class-Wrapper-Parameter weitergegeben werden und Benachrichtigungen dort ausgelöst werden. Ebenso können alle Zugriffe auf die Class-Wrapper-Parameter (aus anderen Teilen des Systems) intern behandelt werden, d.h. an die Interface-Implementierung weitergegeben werden.

Wenn der proprietäre Konfigurationsmechanismus vorsieht, dass die Implementierung des Konfigurations-Interfaces vom Benutzer vorzunehmen ist (oder beispielsweise durch virtuelle Funktionen die Möglichkeit dazu besteht), sind die Möglichkeiten für die Integration stark eingeschränkt. Abbildung 3.14 zeigt die Adapter-Möglichkeit (f) mit einer Monitorklasse.

[16] Benachrichtigungen können verschieden realisiert werden, beispielsweise mit SystemC-Events oder direkten Aufrufen. Callbacks sind zu bevorzugen, da sie geeignet sind, Informationen verzögerungsfrei an einen Beobachter zu leiten.



Compiler/Linker:
Include-/Linker-Verzeichnis:
Parameter-Hersteller-Verzeichnis

Compiler/Linker:
Include-/Linker-Verzeichnis:
Adapter-Verzeichnis

Benutzer-Code:

```
#include "vendor_config_if.h"
struct my_module
: public vendor::config_if {
  void some_method_or_constructor() {
    // Beispiel:
    set_param("name", "val");
  }
};
```

oder

```
#include "vendor_config_if.h"
struct my_module {
  void some_method_or_constructor() {
    // Beispiel:
    vendor_cnf_obj->set_param("name", "val");
  }
};
```

Abbildung 3.13.: PIU-Integration Konfigurations-Interface mit gegebener Implementierung. Links: Originale Interface-Klasse und Implementierung (oben) samt Compiler-Einstellungen und zwei Benutzer-Code-Varianten (unten) des proprietären Modells.
Rechts: Adapter-Variante (e) mit Ersatzinterface und modifizierter Implementierung (oben) und den Auswirkungen auf den Benutzer (unten).
Legende: Die Rechtecke, die Header-Dateien mit Klassendiagrammen darstellen, sind jeweils hinterlegt mit dunkleren Rechtecken, die deren Implementierung darstellen.

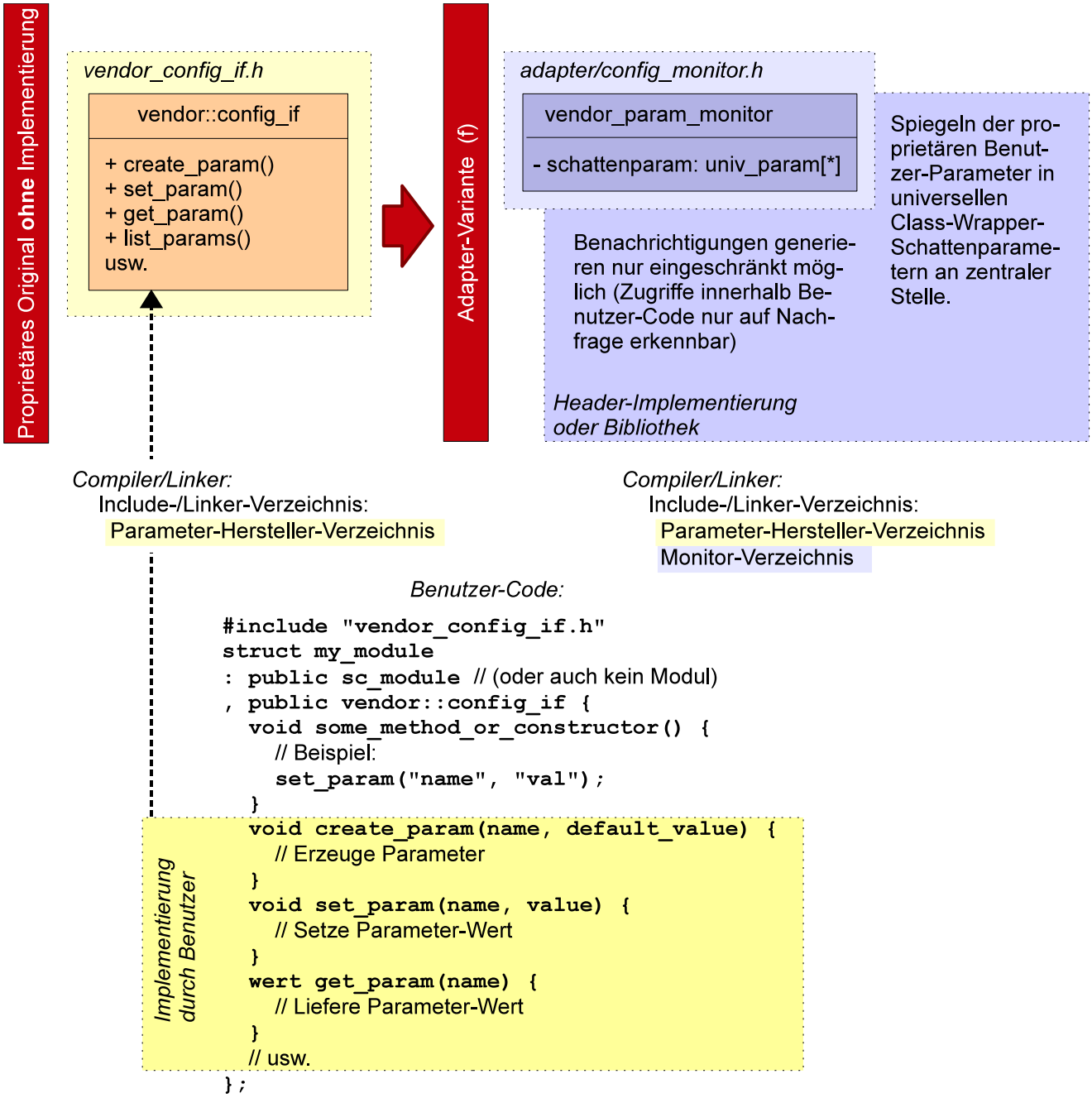


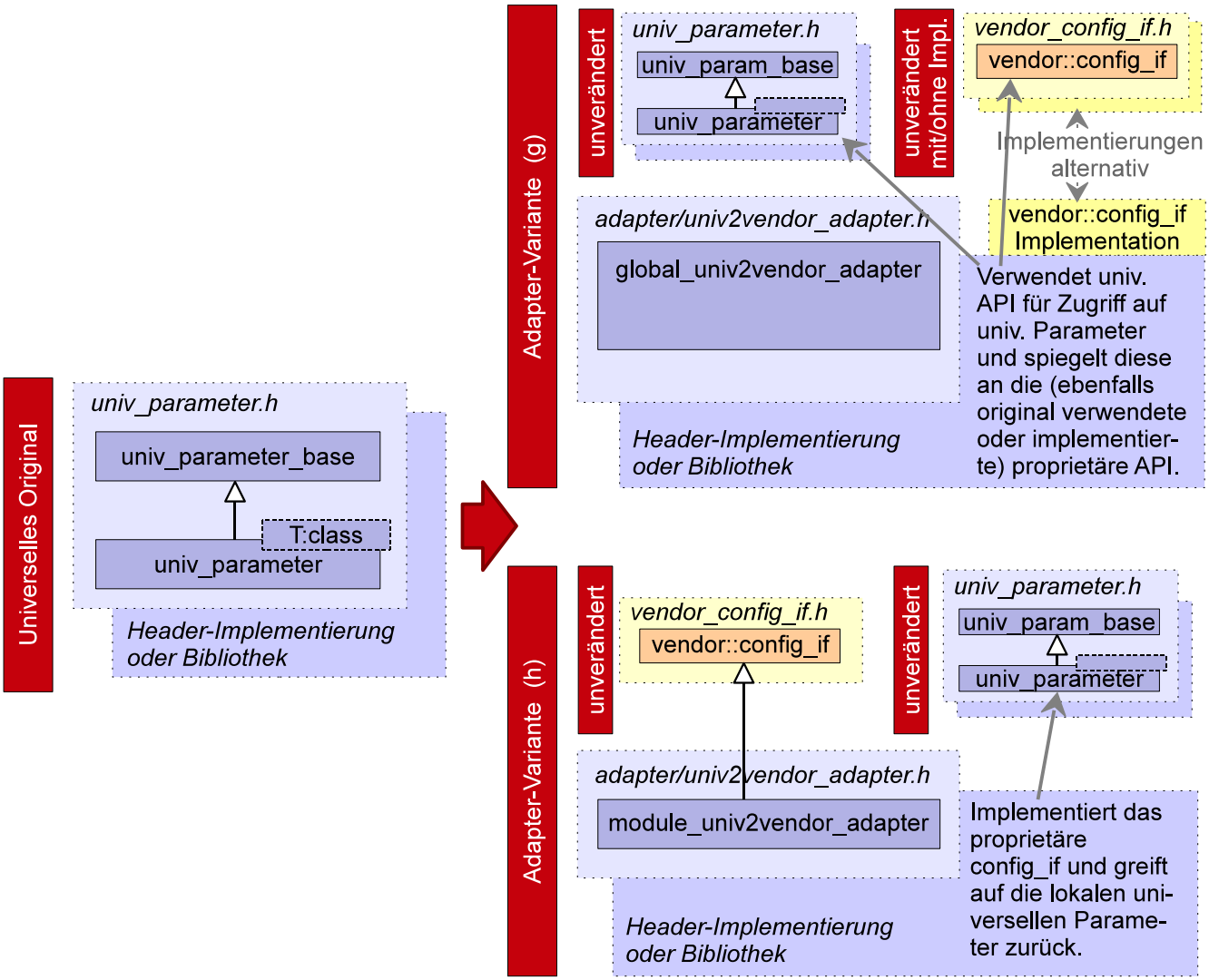
Abbildung 3.14.: PIU-Integration Konfigurations-Interface ohne gegebener Implementierung, d.h. mit Benutzerimplementierung.
Links: Originale Interface-Klasse (oben) samt Compiler-Einstellungen und ein Benutzer-Code-Beispiel (unten) des proprietären Modells oder Konfigurationsmechanismus.
Rechts: Adapter-Variante (f) mit Monitor-Klasse und Implementierung (oben) und den Auswirkungen auf den Benutzer (unten).
Legende: Die Rechtecke, die Header-Dateien mit Klassendiagrammen darstellen, sind jeweils hinterlegt mit dunkleren Rechtecken, die deren Implementierung darstellen.

Da bei dieser Variante der Benutzer die Implementierung vornimmt (in der Abbildung angedeutet durch den Kasten im Benutzer-Code), kann diese nicht von einem Adapter ersetzt werden. Als Folge bleibt nur die Möglichkeit, auf das Konfigurations-Interface selbst in seiner vorgesehenen Art und Weise von außen zuzugreifen. In diesem Beispiel übernimmt das die Implementierung der Monitorklasse `vendor_param_monitor`. Diese Monitorklasse muss im System einmal instanziiert werden und muss die API des proprietären Konfigurationsmechanismus nutzen, um Modell-Parameter aufzufinden und diese in universellen Class-Wrapper-Parametern (Schattenparametern) spiegeln. Falls der proprietäre Mechanismus keine Benachrichtigungen zur Verfügung stellt, können diese bei Zugriffen innerhalb des proprietären Systems auch nicht für die Schattenparameter generiert werden. Eine Synchronisation der Werte und Zustände ist allerdings möglich, da die Schattenparameter vor jedem Lesezugriff eine Benachrichtigung an den Adapter (Monitor) schicken können, der dann aktiv über das proprietäre Interface den Parameterwert und -zustand abfragen kann, um den Schattenparameter zu aktualisieren.

Bei Anwendung der Adapter-Variante (e) wird davon ausgegangen, dass die Implementierung des Konfigurations-Interfaces vom Adapter bereitgestellt wird. Für den Fall, dass der Mechanismus beispielsweise vorsieht, dass der Benutzer die Funktionen überschreibt, können die beiden Adapter-Varianten (e) und (f) kombiniert werden, damit zusätzlich der Adapter-Monitor als Rückversicherung für den möglicherweise überschriebenen Interface-Adapter agieren kann.

UIP-Integration Die allgemeine UIP-Integration für Konfigurations-Interfaces kann aufgrund der unspezifischen Vorgabe für die proprietären Mechanismen nur sehr vage beschrieben werden. Je nach den Regeln des proprietären Konfigurationsmechanismus, in den ein Modell mit universellen Modell-Parametern eingebunden werden soll, variieren die Möglichkeiten und der Aufbau des Adapters.

Abbildung 3.15 soll eine möglichst allgemeingültige Darstellung mit grundlegenden Konzepten verbinden. Der originale Benutzer-Code (linke Seite) verwendet universelle Parameter. Für die Adapter-Variante (g) wird angenommen, dass der proprietäre Konfigurationsmechanismus ein global verfügbares Konfigurations-Interface zur Verfügung stellt (wobei irrelevant ist, ob er selbst eine Implementierung anbietet oder nicht). Der Adapter, (in der Klasse `global_univ2vendor_adapter`) muss nicht im Modell selbst verwendet und instanziiert werden, sondern wird einmal global (zum Beispiel in der systemweiten Testbench) erzeugt. Dieser Adapter verwendet oder implementiert das proprietäre Interface, das entsprechend der Annahme global verfügbar ist und spiegelt die universellen Parameter in dieses Interface. Diese Variante hat den Vorteil, dass der Benutzer-Code (für das Modul) und sogar dessen Compiler-Einstellungen unverändert bleiben können, also nicht einmal ein Neukompilieren erforderlich ist (Binärkompatibiltät). Lediglich die umgebende Simulation (z.B. die Testbench) müssen den Adapter einbinden und verwenden.



Compiler/Linker:
Include-/Linker-Verzeichnis:
Universal-Parameter-Verzeichnis

Benutzer-Code:

```
#include "univ_parameter.h"
struct my_module {
  univ_parameter<int> myparam;
};
```

Compiler/Linker:
Include-/Linker-Verzeichnis:
(g): Universal-Parameter-Verzeichnis
(h): zusätzlich Adapter-Verzeichnis

Benutzer-Code Adapter-Variante (g):

```
#include "univ_parameter.h"
struct my_module {
  univ_parameter<int> myparam;
};
```

Benutzer-Code Adapter-Variante (h):

```
#include "univ_parameter.h"
#include "adapter/univ2vendor_adapter.h"
struct my_module
: public module_univ2vendor_adapter {
  univ_parameter<int> myparam;
};
```

Abbildung 3.15.: UIP-Integration Konfigurations-Interface.
Links: Universelle Parameterklasse (oben) und Compiler-Einstellungen und Modell-Code (unten).
Rechts: Adapter-Variante (g) mit globalem Adapter und Variante (h) mit Moduladapter (oben) und Code-Varianten (unten).
Legende: Die Rechtecke, die Header-Dateien mit Klassendiagrammen darstellen, sind jeweils hinterlegt mit dunkleren Rechtecken, die deren Implementierung darstellen.

Für die Adapter-Variante (h) wird angenommen, dass der proprietäre Konfigurationsmechanismus keine Implementierung für das Konfigurations-Interface bereitstellt und vom Benutzer-Modell erwartet, dass es das Interface implementiert. Aufgrund der Voraussetzungen lässt es sich nicht vermeiden, dass der Benutzer-Code des Modells verändert werden muss: Der Adapter muss eingebunden werden und das Modul muss von der Adapterklasse ableiten, die wiederum von dem proprietären Interface ableitet, um die Anforderungen zu erfüllen. Dadurch muss das Modell auch neu kompiliert werden und zusätzlich zum unveränderten Parameter-Verzeichnis müssen der Adapter-Header und dessen Implementierung bereitgestellt werden. Der Adapter implementiert die Funktionen des Konfigurations-Interfaces und bildet die Aufrufe auf die im Modell verwendeten universellen Parameter ab.

3.3.4. Rangfolgen integrieren

Im Abschnitt 3.1.3 wurden zwei verschiedene Rangfolgevarianten eingeführt, deren Unterschiede erläutert und begründet, warum der hier vorgestellte universelle Konfigurationsmechanismus die zeitliche Konfigurationsrangfolge verwendet. Dieser Abschnitt zeigt grob einige Möglichkeiten der *PIU-Integration* in C++/SystemC[17], mit denen ein proprietärer Mechanismus mit hierarchischer Konfigurationsrangfolge in den universellen Konfigurationsmechanismus integriert werden kann oder mit dessen zeitlicher Rangfolge emuliert werden kann. Dabei werden sowohl die Rangfolgen für die Initialwerte als auch die der sonstigen Wertänderungen betrachtet. Konkrete Realisierungen können im Detail von den hier vorgestellten idealisierten Mustern abweichen, wenn zusätzlich Hierarchien, Namensräume o.ä. die Integration erschweren.

Die Ausgangssituation ist der universelle Konfigurationsmechanismus, der die zeitliche Konfigurationsrangfolge anwendet. Das Ziel ist, die hierarchische Konfigurationsrangfolge nachzubilden, um das Hauptziel dieser Arbeit zu unterstützen, die Austauschbarkeit von Modellen zu erweitern. Das Ziel ist erreicht, wenn ein Modell oder ein Subsystem, das die hierarchische Rangfolge verwendet und aus mehreren hierarchisch gegliederten Modellen besteht, in den universellen Mechanismus eingebunden und konfiguriert werden kann. Dabei soll das eingebundene Modell bzw. Subsystem weiterhin auf die Anwendung einer hierarchischen Rangfolge vertrauen dürfen. Wenn die hierarchische Konfigurationsrangfolge mit der zeitlichen nachgebildet werden kann, sind sowohl Adapter möglich als auch die Anforderungen für einen Standard erfüllt, der beide Rangfolgen unterstützen soll. Die Typen der manipulativen Wertzugriffe können dabei getrennt betrachtet und potentiell mit verschiedenen Ansätzen realisiert werden, da die Rangfolge nur innerhalb der verschiedenen Typen relevant ist.

Die größten Unterschiede und damit Herausforderungen für die Integration treten auf, wenn Modell-Parameter direkt im Konstruktor des besitzenden Moduls verwendet werden. In dem Fall können (wenn mit der zeitlichen Rangfolge die hierarchische nachgebildet werden

[17] Die UIP-Integration wird für die Integration der zeitlichen in die hierarchische Rangfolge nicht gezeigt. Sie ist nicht primäres Ziel dieser Arbeit und ist aufgrund der niedrigen Verbreitung der hierarchischen Rangfolge von untergeordnetem Interesse.

soll) nämlich die Initialwerte vom hierarchisch höheren Modul nicht *nach* dem Konstruieren gesetzt werden, da der Parameterwert dann schon verwendet wurde. Folglich muss der Initialwert *vor* dem Konstruieren des Subsystems gesetzt werden, was zunächst die Gefahr birgt, dass dieser bei der zeitlichen Rangfolge von einem hierarchisch niedrigeren Modul überschrieben wird (vgl. Einführung in Abschnitt 3.1.3). In den anderen Fällen – zumindest wenn der Modell-Parameter zwar im Konstruktor des Subsystems erzeugt, aber nicht verwendet wird – kann der Wert nach der Konstruktion des Subsystems vom hierarchisch höheren Modul verändert werden, sodass er den gewünschten Wert hat, wenn er schließlich vom besitzenden Modul gelesen und verwendet wird.

Im Folgenden werden verschiedene Ansätze für die Rangfolgenintegration vorgestellt. Sie setzen voraus, dass eine Aktivität vor dem Instanziieren des Subsystems durch das hierarchisch höhere Modul durchgeführt werden kann, was praktisch keine große Einschränkung ist. Die verschiedenen Ansätze unterscheiden sich in den Anforderungen, die sie an den Konfigurationsmechanismus stellen, in den die hierarchische Rangfolge integriert werden soll.

Die Rangfolgenintegrationen 1 und 2 sind grundsätzlicher Natur, die dritte ist für die Integration eines existierenden Mechanismus geeignet. Der Vollständigkeit halber wird anschließend eine nicht praktikable Alternative vorgestellt.

Rangfolgenintegration 1: mit Callbacks

Beim ersten Integrationsansatz wird der gewünschte Parameterwert vom in der Modulhierarchie höheren Modul zunächst nicht als Initialwert gesetzt, sondern mit einer sonstigen Wertänderung sofort gesetzt[18], sobald der Parameter vom besitzenden Modul konstruiert wird. Dafür wird der Callback-Mechanismus der API verwendet, der das Modul informiert, wenn ein neuer Parameter konstruiert wird. Damit kein hierarchisch niedrigeres Modul das gleiche Verfahren anwenden kann, also im entsprechenden Callback den Wert wieder überschreiben kann, muss der Parameterwert im Anschluss an die Änderung sofort gegen weitere Änderungen gesperrt werden. Wenn die Integration lediglich den Initialwert betrifft (also keine sonstigen Wertänderungen), kann der Parameter wieder entsperrt werden, nachdem das Subsystem konstruiert ist, da das Subsystem dann keine unerwarteten Wertänderungen des Initialwertes mehr durchführen kann. Wenn der Parameter entsperrt wurde, sind sonstige Wertänderungen anschließend weiterhin möglich. Das wäre auch in der hierarchischen Rangfolge möglich, da es sich nicht mehr um die Änderung eines Initialwertes handeln würde, sondern um eine sonstige Wertänderung. Wenn dagegen auch sonstige Wertänderungen hierarchische Rangfolge haben sollen[19], sollte der Parameter gesperrt bleiben. Wichtig ist in diesem Fall, dass das hierarchisch höchste Modul, das den Parameter setzen möchte und somit Vorrang vor allen anderen beansprucht, den Parameter für alle nicht eigenen manipulativen Wertzugriffe sperrt, sobald es per Callback über die Existenz informiert wurde.

[18] Zugriffstyp: Manipulativer Wertzugriff vom Typ sonstige Wertänderung.
[19] Beispielsweise sind sonstige Wertänderungen bei OVM zwar nicht empfohlen, aber möglich.

Ablauf Im Folgenden wird ein Überblick über die Benutzeraktionen dieses Integrationsverfahrens gegeben:

- Setzen des Initialwerts (optional, s.u.)
- Registrieren eines Callbacks für neue Parameter
- Instanziieren des Subsystems, das hierarchische Rangfolge verwendet bzw. erwartet
- Bei Auftreten eines Callbacks (create_param-Callback) für den gewünschten Parameter
  - Setzen des Wertes (um damit bereits angewendete Initialwerte von anderen Modulen zu überschreiben)
  - und Sperren (lock) des Parameters
- Optional: Entsperren (unlock) des Parameters am Ende des Konstruktors

Voraussetzungen und Anforderungen Dieser Ansatz geht davon aus, dass das Sperren eines Initialwertes nicht möglich ist[20], das Sperren eines expliziten Wertes aber schon. Die Benachrichtigungen (Callbacks) über neu konstruierte Parameter müssen in der Reihenfolge ihrer Registrierung stattfinden, damit das hierarchisch höhere (und zeitlich vorher existierende) Modul seinen Callback eher bekommt, um den neuen Parameter sperren zu können. Ein Modul außerhalb des Subsystems, das den Parameterwert (Initialwert oder als sonstige Wertänderung) setzen möchte, muss den entsprechenden expliziten Parameter sofort sperren, indem es den Callback registriert, sobald es selbst existiert.

Bewertung Das Setzen des Initialwerts ist wirkungslos (höchstens für die Debug-Analyse interessant), da der Wert in der Callback-Funktion gesetzt werden muss. Das Verfahren ist nicht sonderlich übersichtlich für den Benutzer und damit eine unsaubere Lösung unter Umgehung des vorgesehenen Setzens von Initialwerten. Allerdings benötigt dieses Verfahren weniger Merkmale im Konfigurationsmechanismus, dessen Anforderungen beim folgenden Verfahren höher sind. Um eine durchgängige hierarchische Rangfolge zu erreichen, müssen sich alle Modelle, auch im Subsystem, an dieses Verfahren halten. Dieser Ansatz hat zu viele Nachteile und wird deswegen nicht weiter betrachtet.

Rangfolgenintegration 2: mit Sperren

Der zweite Integrationsansatz verwendet ein zusätzliches Merkmal, das der zugrunde liegende Konfigurationsmechanismus anbieten muss: Ein einmal gesetzter Initialwert wird vor einem späteren Überschreiben, z.B. durch hierarchisch tiefer liegende Module, geschützt. So kann ein hierarchisch höheres Modul zunächst den Initialwert setzen und sperren, bevor das unterliegende Subsystem erzeugt wird. Als Folge kann der Initialwert von keiner Instanz des

[20] Andernfalls wäre die Rangfolgenintegration 2 eleganter.

Subsystems überschrieben werden. Im Unterschied zur Rangfolgenintegration 1 wird hier der Initialwert gesperrt, nicht der Parameterwert. Ein Entsperren des Initialwerts ist nicht notwendig, da dieser nach dem Erzeugen des Parameters durch das besitzende Modul nicht mehr von Bedeutung ist und die Sperrung dann keinen Einfluss mehr auf den Wert des Parameters hat.

Für die hierarchische Rangfolge sonstiger Wertänderungen kann das Modul das Verfahren benutzen, das in der Rangfolgenintegration 1 für das Setzen des Initialwerts verwendet wird: Der Parameter wird vom hierarchisch höchsten – und damit zeitlich am ehesten existierenden – Modul gesperrt, sobald ein Callback dessen Existenz bekannt gibt.

Ablauf Im Folgenden wird ein Überblick über die Benutzeraktionen dieses Integrationsverfahrens (für hierarchische Rangfolge von Initialwerten) gegeben:

- Setzen des Initialwerts mit anschließendem (manuellem oder automatischem) Sperren des Initialwerts
- Instanziieren des Subsystems

Voraussetzungen und Anforderungen Die API des Konfigurationsmechanismus muss das Sperren von Initialwerten unterstützen[21]. Für die hierarchische Rangfolge von sonstigen Wertänderungen muss auch das Sperren von Parameterwerten expliziter Parameter unterstützt werden.

Bewertung Dieses Integrationsverfahren behält die Bedeutung des gesetzten Initialwertes bei und erlaubt es ohne Probleme, im Subsystem Module zu verwenden, die sich dieser Integration nicht bewusst sind. Zudem sind die Anforderungen an Code-Änderungen für das hierarchisch höhere Modul (und damit die Anforderungen an den Benutzer) minimal, da lediglich das Sperren des Initialwertes vorgenommen werden muss und die Regel beachtet werden muss, dass der Initialwert vor Erzeugung des Subsystems gesetzt werden muss – was aber sowieso der Fall sein sollte.

In Abschnitt 6.4 wird eine konkrete Realisierung gezeigt.

Rangfolgenintegration 3: Original verwenden

Die beiden vorangehenden Rangfolgenintegrationen beschreiben Verfahren, mit denen das Verhalten von hierarchischer Rangfolge in einem die zeitliche Rangfolge verwendenden Mechanismus nachgebildet werden kann. Anwendungsfälle können einerseits extra für diesen Zweck geschriebene Modelle sein, andererseits Adapter zu anderen (z.B. bereits existierenden) Konfigurationsmechanismen, die die hierarchische Rangfolge verwenden. Für den letzteren Fall wird hier ein weiterer Ansatz präsentiert:

[21] Entweder muss es einen manuellen Aufruf geben, oder die API muss das automatisch durchführen, wobei dies der API entweder durch eine zusätzliche Funktion oder einen (u.U. optionalen) Funktionsparameter durch den Benutzer mitgeteilt werden muss.

Ein bereits existierender Konfigurationsmechanismus mit hierarchischer Rangfolge kann möglicherweise so geändert werden, dass er intern Parameter des universellen Mechanismus verwendet. Die Voraussetzung ist, dass der Quellcode des Mechanismus zum Modifizieren verfügbar ist. Dieser Ansatz filtert mit dem existierenden Mechanismus jeden Parameterzugriff durch Modelle, die die originale API verwenden. Damit verhindert er den direkten Zugriff auf die internen Parameter des universellen Mechanismus. Folglich kontrolliert der originale Mechanismus weiterhin die Rangfolge. Die intern verwendeten Parameter spiegeln dann jeden tatsächlich Zugriff wider. Lediglich im Zusammenspiel mit der Konfiguration durch fremde Modelle oder Testbenches in einer gemischten Simulation müssen die Rangfolgen entweder manuell bedacht werden oder mit einer der anderen Rangfolgenintegrationen gelöst werden, da diese die originale API umgehen würden.

Die OVM-Integration in Abschnitt 6.5 ist eine konkrete Realisierung dieser Variante.

Alternative

Zu guter Letzt ist es auch möglich, das Verhalten hierarchischer Rangfolge ohne weitere Unterstützung durch den Konfigurationsmechanismus nachzubilden, indem der Benutzer vor jedem Setzen eines Initialwertes zunächst prüft, ob dieser bereits gesetzt wurde und das Setzen nur durchführt, wenn das nicht der Fall ist. Die Alternative bürdet dem Benutzer eine Verantwortung auf, erzeugt damit Fehlerquellen und ist höchst unzuverlässig. Zudem verhindert es die Wiederverwendbarkeit von Modellen, die für den Einsatz in einem Mechanismus mit zeitlicher Rangfolge geschrieben wurden, da diese sich nicht an die Regeln halten.

3.3.5. Fazit und Einschränkungen

Die vorausgehenden Abschnitte führen in die technischen Möglichkeiten der Integration verschiedener Konfigurationsmechanismen ein. Unterschiede zwischen den Mechanismen können zum Teil überwunden werden. Das Design des universellen Mechanismus erlaubt vor allem die Integration von proprietären Mechanismen. Die andere Richtung ist im Wesentlichen beschränkt durch die unterstützten Merkmale des jeweiligen proprietären Mechanismus. Bestimmte Merkmale, die ein Adapter nicht transparent hinzufügen kann, können dem universellen Modell nicht zur Verfügung gestellt werden, wenn sie vom proprietären Mechanismus oder der IDE nicht unterstützt werden.

Es gibt des Weiteren Mechanismen, die eine Integration von fremden (z.B. des universellen) Mechanismen kaum erlauben, da sie sehr starke und unflexible Annahmen über den Konfigurationsmechanismus treffen. Ein fremder Mechanismus kann beispielsweise schwer in eine IDE integriert werden, wenn sie naives Quellcode-Parsing betreibt und die Struktur im Quellcode des fremden Mechanismus sich grundlegend vom Zielmechanismus unterscheidet. Hier könnten beispielsweise Präprozessor-Makros Abhilfe schaffen, die jedoch problematisch sind, wenn sie wie normaler Code aussehen (weil sie solchen ersetzen sollen bzw. für den Parser wie solcher aussehen sollen) und sich der Namensräume nicht bewusst sind.

4. Modell-Middleware

Inhalt

In diesem Kapitel wird die Basis für den in dieser Arbeit entwickelten universellen Konfigurationsmechanismus gebildet. Da es sich dabei nicht nur um einen einzelnen Konfigurationsmechanismus handelt, sondern aufgrund flexibler Schnittstellen um verschiedene Mechanismen, spreche ich von einem **Konfigurationsframework**, das auf dem in diesem Kapitel eingeführten **Middlewareframework** aufsetzt.

4.1. Konzept

Für dieses Kapitel sind zunächst ausgewählte Anforderungen aus Abschnitt 3.2 von Bedeutung: Es sollen verschiedene Konfigurationsansätze integrierbar sein (AA3). Da für diese verschiedenen Mechanismen nicht festgelegt werden kann, wo diese ihre jeweiligen Modell-Parameter speichern, ist es sinnvoll, eine zentrale Datenbank über alle im Konfigurationsframework vorhandenen Parameter zu verwalten[1]. Das wird deutlicher in den Anforderungen, ein globales Verzeichnis anbieten zu können (KA7) oder auf jeden Parameter zugreifen zu können (KA8). Die wichtigste Anforderung, die für die im Folgenden vorgestellte Middleware-Lösung spricht, ist die geforderte flexible interne API für die verschiedenen Adapter zu unterschiedlichen Konfigurationsmechanismen (AA5) und die Notwendigkeit, die interne Konfigurations-API um nicht bedachte Merkmale erweitern zu können (AA4), jeweils ohne bereits bestehende Modelle neu kompilieren zu müssen. Des Weiteren muss darauf geachtet werden, dass der Mechanismus möglichst unauffällig für SystemC-Analysen ist (KA15).

Als zusätzliche Anforderung soll die zu entwickelnde Middleware nicht nur für die Konfiguration verwendet, sondern auch für weitere Meta-Funktionalität verwendet werden können.

[1] Es ist der Implementierung überlassen, ob die zentrale Datenbank tatsächlich als Datenstruktur geführt wird oder bei jedem Zugriff dynamisch erstellt wird.

Um ein monolithisches Konfigurationsframework zu entwerfen, ist ein Middleware-Ansatz zunächst nicht notwendig. Ein geradliniger Ansatz wäre eine Klasse mit einer mächtigen API für sämtliche Konfigurationsmerkmale der Anforderungen. Adapter könnten Verbindungen zu proprietären Mechanismen herstellen. Dieser einfache und offensichtliche Ansatz hat allerdings verschiedene Nachteile:

Der unflexible Ansatz führt zu Problemen, wenn dieser Konfigurationsmechanismus mit seiner Parameterdatenbank beispielsweise an ein Werkzeug eines anderen Herstellers angeschlossen werden soll, das eine eigene Parameterdatenbank verwaltet. Die monolithische Klasse müsste möglicherweise stark modifiziert werden, um die Parameterdatenbank entweder zu spiegeln (einen Mirror zu verwalten) oder komplett auf eine eigene Verwaltung zu verzichten und auf die des externen Werkzeugs zurückzugreifen.

Ein gravierenderes Problem tritt auf, wenn ein neuer Konfigurationsmechanismus mit bisher unbeachteten Merkmalen unterstützt werden soll. In dem Fall muss die monolithische API samt der Implementierung angepasst werden. Das ist möglich, führt aber dazu, dass sämtliche bestehenden Modelle zumindest neu kompiliert werden müssen und möglicherweise sogar deren Quellcode angepasst werden muss, um zusammen mit dem modifizierten Konfigurationsframework in einer Simulation verwendet werden zu können. Mit diesem Ansatz würde die Abwärtskompatibilität eingeschränkt und von kompilierten Modellen sogar ausgeschlossen werden. Es ist also sinnvoll und wünschenswert, einen flexibleren Ansatz zu finden.

Flexibler Ansatz

Für einen flexiblen Ansatz wird die Implementierung des Konfigurationsmechanismus von der API getrennt, also eine zusätzliche Trennungsebene zwischen der API des Endnutzers und der Implementierung des Konfigurationsmechanismus eingeführt. Dieser Abschnitt diskutiert zwei verschiedene Ansätze einer solchen Trennung: Ein aus Sicht eines Softwareentwurfs naheliegender Ansatz mit einer Trennungsebene und ein weitergehender Trennungsansatz mit zwei Trennungsebenen.

Als Anforderung für beide Ansätze muss eine flexible API um Merkmale erweitert werden können, ohne dass bestehende Modelle neu übersetzt werden müssen, weil sich die Header-Dateien oder die Implementierung ihrer API geändert haben. Dazu ist es notwendig, die Bereitstellung der Funktionalität von der API zu trennen. Des Weiteren ist es von Vorteil, wenn der Ansatz auch für weitere Meta-Funktionalität elegant wiederverwendet werden kann.

Der aus Softwareentwurfsicht zunächst naheliegende Ansatz in C++ ist die Verwendung von Interface-Klassen (Abbildung 4.1). Die Funktionalität wird von einer Hauptklasse implementiert, die von einer abstrakten Interface-Klasse StdInterface ableitet. Auf Anwenderseite sollte sich aufgrund der variablen Konfigurationsmechanismen eine Zugriffshilfsklasse befinden, die eventuell notwendige Konvertierungen verbirgt und eine API für den Endnutzer bereitstellt (StdZugriffsklasseA und StdZugriffsklasseB), die sich potentiell von der Interface-Klasse unterscheidet. Diese Zugriffsklasse bekommt einen Pointer vom Typ des

Interfaces auf das Objekt der Hauptklasse, mit dem die Zugriffsklasse arbeiten kann. Soll neue Funktionalität hinzugefügt werden, kann die Hauptklasse diese implementieren und von einem zusätzlichen abstrakten Erweiterungsinterface ErwInterface ableiten. Eine neue Zugriffsklasse ErwZugriffsklasse kann mit diesem neuen Pointertyp arbeiten. Die ursprüngliche Zugriffsklasse kann nach wie vor mit dem ursprünglichen Interface auf das Objekt der Hauptklasse zugreifen.

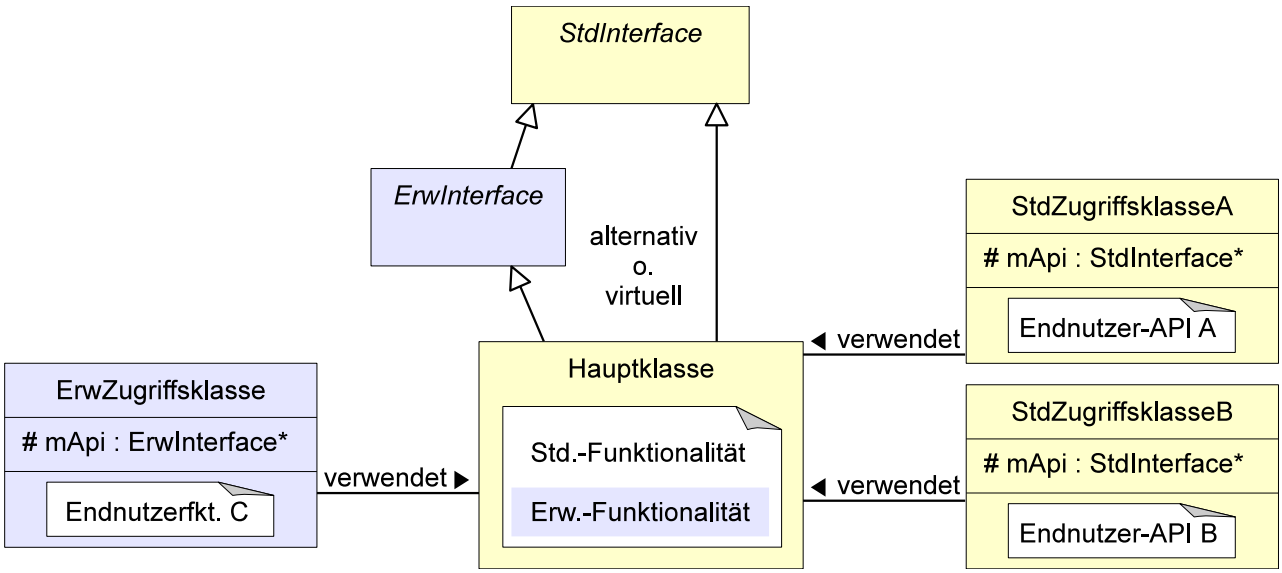


Abbildung 4.1.: Mögliche Realisierung mit einer Trennungsebene

Der weitergehende Ansatz mit zwei Trennungsebenen (Abbildung 4.2 auf Seite 69) transportiert Aufrufe zwischen Zugriffsklassen User_API_xyz und der Hauptklasse Konfigurations_Service_Plugin über erweiterbare Transportcontainer (Transaktionen). Als zusätzliches Element im System gibt es ein internes Vermittlerobjekt (Core), das als zweite Trennungsebene eine interne statische Schnittstelle Int_API besitzt. Es nimmt Transaktionen entgegen und leitet sie an das Hauptobjekt weiter. Der Transportcontainer trägt Parameter und kann dynamisch erweitert werden, ohne dass sich die statische Schnittstelle ändert. Somit können erweiterte und nicht erweiterte Zugriffsklassen stets das gleiche interne Interface verwenden. Die Hauptklasse kann erweitert werden, indem sie neue Erweiterungen in Transportcontainern nutzt. Nur erweiterte Zugriffsklassen schicken erweiterte Transaktionen, die Hauptklasse empfängt beide. Abbildung 4.2 zeigt die Zusammenhänge anhand des ausgewählten Middleware-Konzepts.

Der vorgestellte Ansatz mit zwei Trennungsebenen benötigt eine Art Routing der Transaktionen im Vermittlerobjekt. Als Folge bedeutet es weder mehr Implementierungsaufwand, noch schlechtere Performance, weitere Funktionalitäten mit jeweils einer Hauptklasse und mehreren Zugriffsklassen zu erlauben. Deswegen begünstigt dieser Ansatz den Einsatz weiterer Meta-Funktionalität und ist nicht auf Konfiguration beschränkt. Dieser Ansatz kann aufgrund seines Aufbaus als Middleware verstanden werden.

Zunächst sind die beiden Ansätze bezüglich der technischen Möglichkeiten äquivalent. Mit beiden Ansätzen lässt sich Erweiterbarkeit und dabei Binärkompatibilität realisieren. Der Ansatz mit zwei Trennungsebenen wirkt jedoch eleganter, generischer und variabler und bietet zudem Möglichkeiten zum Debuggen, was anhand der im Folgenden aufgezeigten Vor- und Nachteile dieses Ansatzes gezeigt werden soll:

Nachteil:

- Die zusätzliche Trennungsebene, also die Datenübertragung über Transaktionen, wirkt sich nachteilhaft auf die Performance aus: Parameter müssen in die Attribute von Transaktionen kopiert werden und Transaktionen müssen vermittelt werden.

Vorteile:

- Dieser Middleware-Ansatz erlaubt Topologieänderungen zur Laufzeit. Verschiedene Meta-Funktionalitäten können dynamisch hinzugefügt und entfernt werden. Das beinhaltet auch das dynamische Hinzufügen von Elementen zum Debuggen.

- Als zentraler Ansatzpunkt kann der Vermittler zum Debuggen und zur Analyse verwendet werden.

- Durch Wiederverwendbarkeit bei mehreren Meta-Funktionalitäten vereinfacht ein gemeinsamer Mechanismus die Wartbarkeit gegenüber Einzellösungen.

- Transaktionen bieten eine elegante Möglichkeit, den Aufrufer als Absender zu identifizieren. Das kann einerseits für die Kommunikation zwischen Zugriffsklassen und Hauptklassen und andererseits zum Debuggen verwendet werden.

- Zum Debuggen kann ein generischer, von der Meta-Funktionalität unabhängiger Logger verwendet werden. Das verringert den Entwicklungsaufwand für neue Funktionalitäten.

- Bei Verwendung dieses Ansatzes ist es möglich, die Hauptklasse weniger erweitert zu verwenden als beispielsweise einige der Zugriffsklassen. Das ermöglicht Freiheiten bei der Auswahl der Hauptklasse, wenn einige erweiterte Merkmale nicht notwendig oder verfügbar sind.

- Ist ein adressiertes Ziel nicht verfügbar (z.B. weil das Hauptobjekt fehlt), kann dieser Fehler im zentralen Vermittlerobjekt behandelt werden und vereinfacht somit die Entwicklung der Zugriffsklassen.

Neutrale Eigenschaften:

- Es sind keine virtuellen Funktionsaufrufe notwendig, wodurch sich im Vergleich zum Ansatz mit einer Trennungsebe ein kleiner Performancevorteil ergeben könnte, der durch den Nachteil der zusätzlichen Trennungsebene aber unbedeutend wird.

- Der Erweiterungsmechanismus der Transaktionen hat keine negativen Auswirkungen auf die Performance.

Die vielen Vorteile des Ansatzes mit zwei Trennungsebenen sind nicht-technischer Art, vereinfachen aber den Umgang und machen ihn generischer für weitere Meta-Funktionalitäten als nur Konfiguration. Deswegen ist der für das universelle Konfigurationsframework verwendete Ansatz die im Folgenden vorgestellte Modell-Middleware GreenControl, die zwei Trennungsebenen verwendet.

Grundlegendes Konzept

Abbildung 4.2 zeigt das vereinfachte grundlegende Konzept der Middleware am Beispiel der Konfiguration. Im Folgenden wird das Konzept anhand der Abbildung erläutert. Der obere Teil zeigt die herkömmliche Simulation des Real-Modells, in diesem Beispiel drei über einen Bus verbundene IP-Modelle. Die verwendeten Begriffe sind zunächst selbsterklärend und werden beim detaillierten Konzept auf der nächsten Seite abschließend definiert. Um ihre Konfigurierbarkeit herzustellen, verwenden die Modelle als Zugriffsklasse jeweils eine Endbenutzer-User-API. IP_1 und IP_2 verwenden die gleiche User_API_A, Modell IP_3 verwendet eine um ein zusätzliches Merkmal erweiterte API, die User_API_extended_B. Die User-APIs

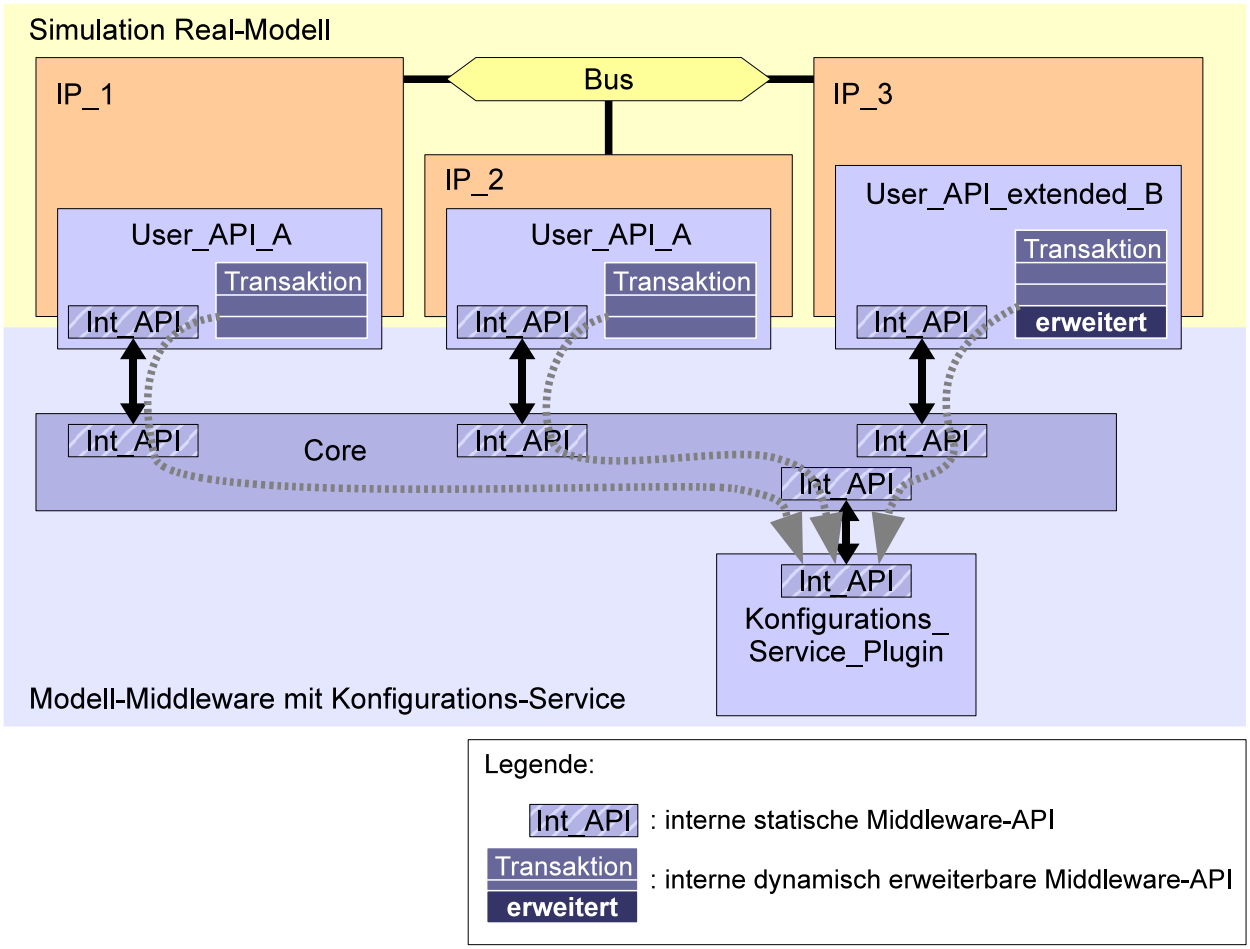


Abbildung 4.2.: Grundlegendes Middleware-Konzept mit zwei Trennungsebenen

verbinden sich (automatisch) mit dem Vermittlerobjekt der Middleware, dem Core. Die Konfigurationsfunktionalität wird von der Hauptklasse, dem Konfigurations-Service-Plug-in bereitgestellt, das sich ebenfalls mit dem Core verbindet. Der Core ist eine Art Router, der Verbindungen von User-APIs und Plug-ins entgegennimmt und ein Singleton ist. Die Verbindung mit dem Core funktioniert über eine interne statische API (in der Abbildung schraffiert dargestellt): Die Funktionen dieser internen API nehmen Transaktionen festgelegten Typs entgegen und der Core leitet sie an das gewünschte Ziel (hier das Konfigurations-Service-Plug-in) weiter (vgl. helle unterbrochene Pfeile in der Abbildung). Die Transaktionen sind dynamisch erweiterbar, sodass der Core und die interne API nicht geändert werden müssen, wenn die Transaktionen (und damit auch optional eine Auswahl von User-APIs und das Plug-in) mit neuen Merkmalen ausgestattet werden. Das Service-Plug-in muss lediglich Abwärtskompatibilität bewahren, indem es weiterhin nicht erweitere Transaktionen und alle Arten von erweiterten Transaktionen, die von einer der User-APIs gesendet werden könnten, empfangen und korrekt behandeln kann. Der umgekehrte Weg einer Transaktion ist in der Abbildung durch beidseitige Pfeile angedeutet: Auch das Service-Plug-in und die User-APIs können Transaktionen an User-APIs senden.

GreenControl-Middleware-Konzept

Auf Basis des grundlegenden Konzepts und der Erweiterung auf beliebige Meta-Funktionalitäten wird in diesem Abschnitt der Aufbau der Middleware GreenControl beschrieben, die die Basis für das in dieser Arbeit vorgestellte Konfigurationsframework ist.

Dieser Ansatz ist von mir im GreenSocs-Projekt GreenControl realisiert worden. Ein Abriss eines frühen Status des Projekts ist in [Klin08] sehr kurz beschrieben. Im Unterschied dazu wird von mir in dieser Arbeit für die Kommunikation kein TLM-Framework verwendet, das eigentlich für die Kommunikationsmodellierung des Real-Modells vorgesehen ist. Damit wird die Sichtbarkeit in SystemC-Analysen, wie in Anforderung KA15 gefordert, minimiert und Performance-Einbußen durch eine optimierte Implementierung vermieden. Erste Ergebnisse sind in [SKG+09] erschienen.

Die Middleware **GreenControl** besteht aus den folgenden Elementen:

- **GreenControl-Service**: Ein GreenControl-Service ist die Menge aller in den Unterpunkten vorgestellten Elemente, die einem gemeinsamen Zweck (z.B. Konfiguration oder Analyse) dienen.
  - **GreenControl-Service-Plug-in**: Die Implementierung der eigentlichen Funktionalität wird im GreenControl-Service-Plug-in (kurz: Plug-in) realisiert. Für einen Service gibt es genau ein Service-Plug-in, das folglich ein Singleton-Objekt ist.
  - **GreenControl-User-API**: Das vom Endnutzer zu verwendende Objekt zum Zugriff auf den Service wird GreenControl-User-API (kurz: User-API) genannt.

Eine User-API bietet dem Benutzer eine Schnittstelle und enthält eine einfache Implementierung für die Weiterleitung der Anfrage zum Service-Plug-in oder enthält eine aufwändigere Implementierung, die Konvertierungen vornimmt, bevor die Anfrage an das Service-Plug-in gesendet werden kann. Für ein Service-Plug-in kann es verschiedene User-APIs geben, die verschiedene Schnittstellen für den gleichen Service anbieten.

- **GreenControl-Core**: Zwischen den Service-Plug-ins und User-APIs wird eine Bus-ähnliche Kommunikations-Struktur geschaltet. Dessen API (das `gc_port_if`-Interface) nimmt eine **GreenControl-Transaktion** entgegen und ist somit zunächst statisch. Die Transaktion kann aber dynamisch erweitert werden, was es beliebig vielen, potentiell verschiedenen User-APIs erlaubt, sich mit dem Service-Plug-in zu verbinden[2]. Parameter und Rückgabewerte transportiert die Transaktion in ihren Datenstrukturen (Attributen).

Abbildung 4.3 auf der nächsten Seite zeigt das Konzept mit zwei Services. Drei IP-Modelle verwenden verschiedene GreenControl-User-APIs, um auf zwei Services (A und B) zuzugreifen. Alle User-APIs und Service-Plug-ins sind an den Core angeschlossen und senden und empfangen Transaktionen. Die User_API_A versendet einfache Transaktionen an das Service_Plugin_A, User_API_B versendet ebenfalls einfache Transaktionen an das zugehörige Service_Plugin_B. Die in IP_2 verwendete User_API_A1 versendet erweiterte Transaktionen an das Plug-in A, die wohlgemerkt über den gleichen Port mit identischem Interface angeschlossen sind. Die User_API_B1 erweitert die Transaktionen um weitere Felder und schickt sie an das Plug-in B. Die beiden Service-Plug-ins müssen in der Lage sein, sowohl die jeweils für sie bestimmten nicht erweiterten als auch die erweiterten Transaktionen empfangen und bearbeiten zu können. Hier wird deutlich, dass beispielsweise die User_API_B1 samt dem Modell IP_3 entwickelt werden kann, nachdem die anderen User-APIs und Modelle bereits existieren und sogar bereits kompiliert sind. Da der Core die Verbindungen unabhängig von den verwendeten Erweiterungen der Transaktionen herstellt, muss lediglich das Service-Plug-in angepasst werden.

Eine **Modell-Middleware** ist eine Kommunikationsstruktur für verschiedene Werkzeuge für den Entwurf von SystemC-Modellen. Die Modell-Middleware besitzt eine statische API, die dynamisch erweitert werden kann.

Ein Hauptziel dieser Arbeit ist es, die Austauschbarkeit von SystemC-Modellen zu verbessern. Folglich soll auch die Modell-Middleware für SystemC geeignet sein und darf auch darauf zugeschnitten sein. Es sollte allerdings auf eine möglichst geringe Abhängigkeit und Verwendung von SystemC geachtet werden, damit die Middleware selbst, die als Teil der Simulation kompiliert wird, im SystemC-Simulator nicht oder möglichst wenig in Erscheinung tritt (Anforderung KA15, Unauffälligkeit in SystemC). Damit die Middleware für den

[2]Das Transaktionskonzept ist angelehnt an die erweiterbaren Transaktionen von TLM-2.0.

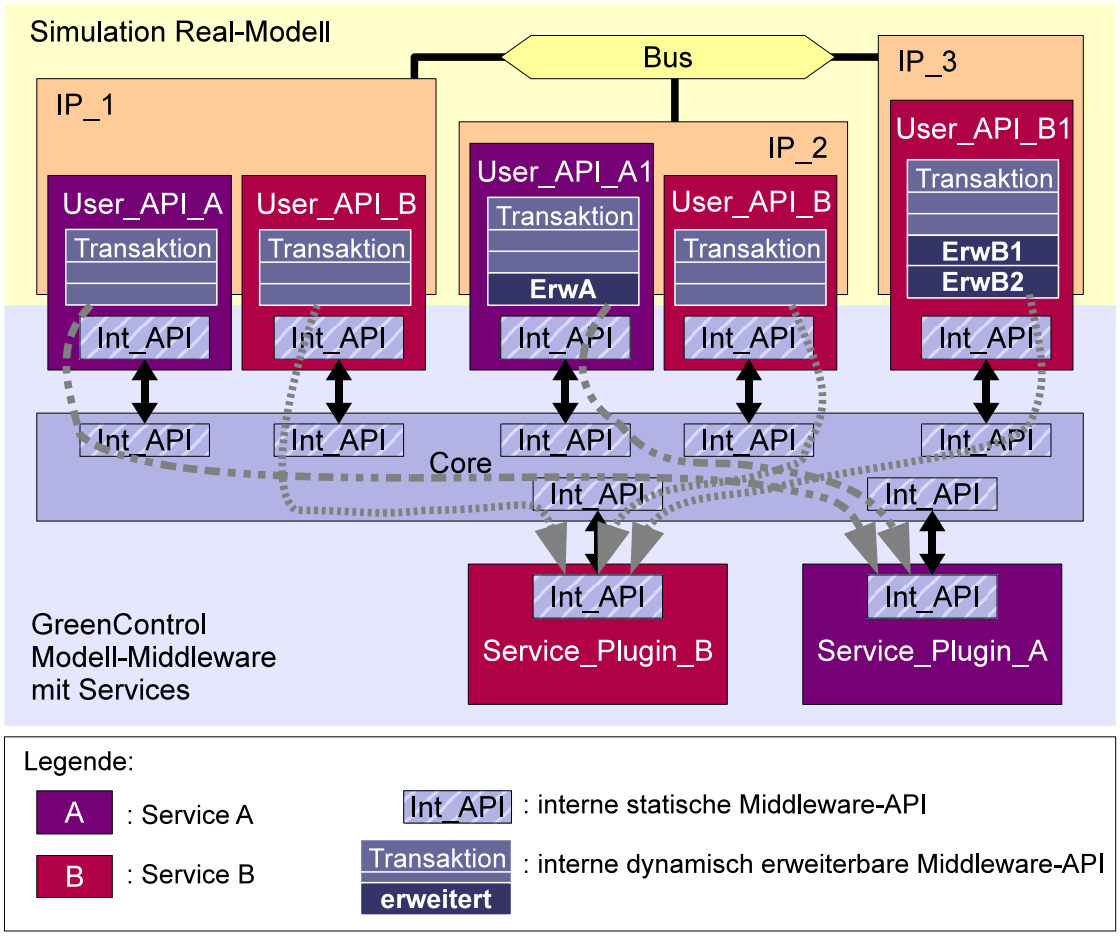


Abbildung 4.3.: GreenControl Middleware-Konzept

Endnutzer transparent ist, sollte es beispielsweise vermieden werden, dass Analysetools[3] die Existenz von GreenControl und der Services bemerken, wenn sie die Modulhierarchie oder andere SystemC-Objekte durchsuchen.

Zusammenfassend ist GreenControl eine Modell-Middleware und stellt einen einfachen Verbindungsmechanismus samt einer schlanken Kommunikationsschnittstelle zur Verfügung. Ein Service-Plug-in implementiert die Funktionalität und potentiell mehrere User-APIs stellen möglicherweise umfangreiche Schnittstellen bereit und übertragen sie durch die schlanke Middleware zum Plug-in. Die in der GreenControl-Plattform vereinigten Services sollen eine Entwicklungsumgebung für SystemC-Modelle ergeben, die Funktionen wie Konfiguration, Untersuchung, Analyse und Debugging anbietet.

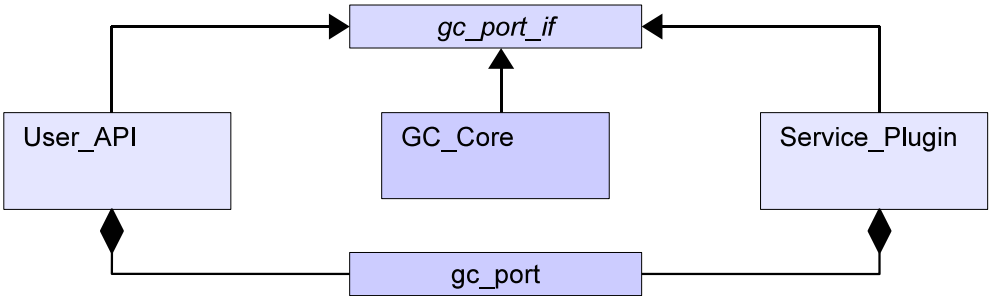
4.2. Aufbau und internes Interface

Die Modell-Middleware soll ein statisches Interface anbieten, das bei Anpassungen der Services weder geändert noch die evtl. als Bibliothek vorhandene Middleware-Implementierung neu kompiliert werden muss. Trotzdem sollen die übertragenen Anfragen flexibel erweiterbar

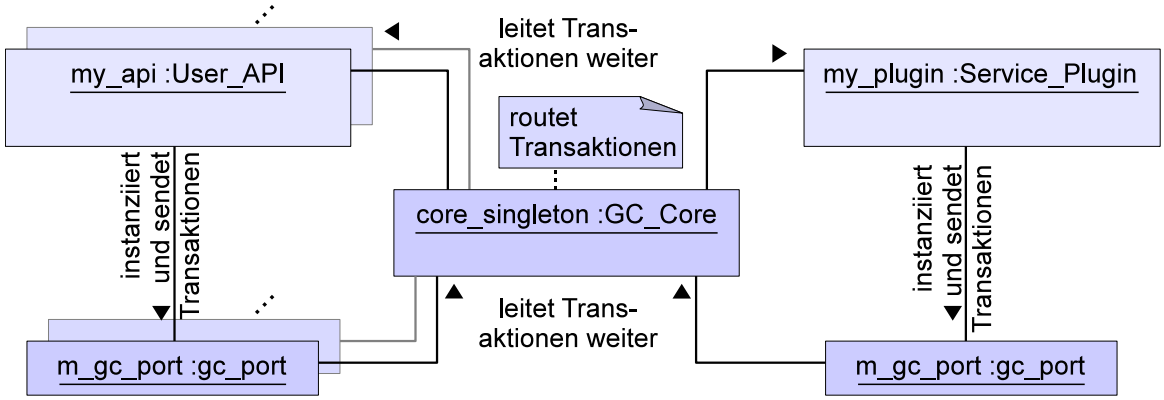
[3] Anaylsetools können entweder Teil der GreenControl-Plattform oder andere Tools sein, in denen eine Simulation mit GreenControl anaylsiert wird.

sein. Um die Anforderungen an den hier entwickelten universellen Konfigurationsmechanismus zu erfüllen, verwendet diese Middleware einen Transaktionsansatz, der in ähnlicher Form auch in OSCI TLM-2.0 zum Einsatz kommt. Dieses aus Benutzersicht interne Interface wird zusammen mit Details des Aufbaus der Modell-Middleware GreenControl in diesem Abschnitt ausführlich beschrieben.

Abbildung 4.4(a) zeigt ein Übersichts-Klassendiagramm des Cores und der beteiligten Kommunikationskomponenten. Die Klasse `gc_port` ist ein von der Middleware bereitgestellter Port und kann von den User-APIs und Service-Plug-ins verwendet werden. Der Core und die User-APIs und Service-Plug-ins müssen das Interface `gc_port_if` implementieren, um jeweils Transaktionen von einem Port oder dem Core (der die in den Ports bereitgestellte Funktionalität selbst implementiert) entgegen nehmen zu können. Abbildung 4.5 zeigt ein detailreicheres Klassendiagramm. In beiden Diagrammen sind die Middleware-Komponenten dunkler gefärbt, die Service-abhängigen Komponenten heller. Die im oberen Teil von Abbildung 4.5 schraffierten Klassen gehören zu dem in Abschnitt 4.3 beschriebenen Erweiterungsmechanismus. Zusammenhänge zwischen den Klassen macht das Objektdiagramm in Abbildung 4.4(b) deutlich. Im Folgenden werden die dargestellten Klassen ausführlicher beschrieben.



(a) Übersicht (Klassendiagramm)



(b) Beispiel (Objektdiagramm)

Abbildung 4.4.: Middleware-Core

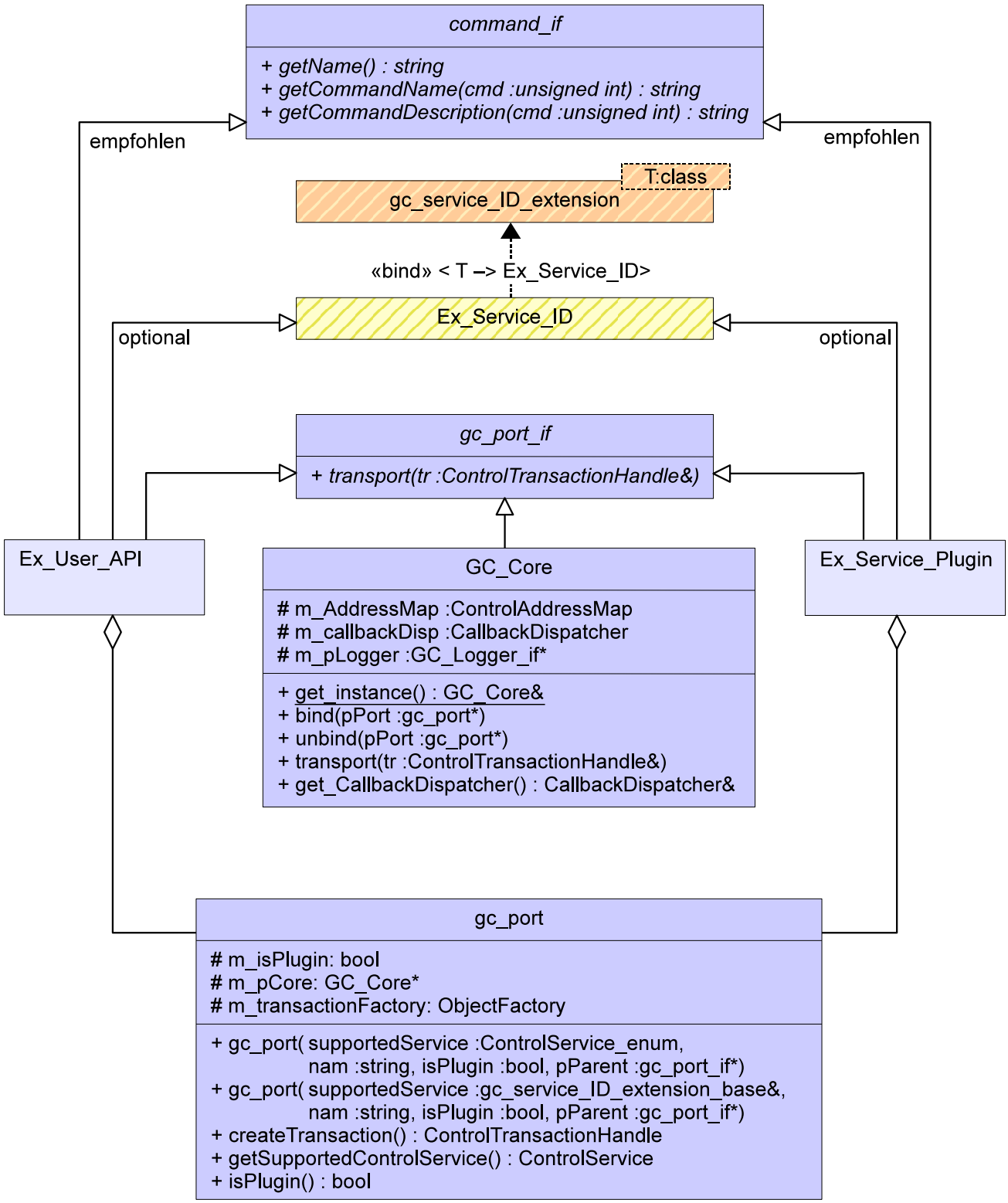


Abbildung 4.5.: Middleware-Core-Details mit Beispiel-Service Ex (Klassendiagramm)

Ein *Port* (Klasse `gc_port`) soll von jeder User-API und jedem Service-Plug-in instanziiert werden. Er stellt im Wesentlichen zwei Convenience[4]-Funktionalitäten bereit: Zum einen verbindet er sich während dem Konstruieren automatisch mit dem Core und verwaltet zum anderen eine Transaktions-Factory[5]. Die in der Funktion `createTransaction` bereitgestellte Factory wird im Memory-Management der Transaktionen in Abschnitt 4.2.2 detailliert vorgestellt. Im Konstruktor wird angegeben, welcher Service von der User-API angeboten und adressiert bzw. vom Plug-in angeboten wird, welchen Namen die User-API bzw das Plug-in hat und ob der Besitzer des Ports ein Plug-in ist. Zudem wird der Pointer zum Besitzer in Form des `gc_port_if`-Interfaces übergeben, damit der Port diesen als Empfänger für den Rückkanal beim Core registrieren kann.

Das *Port-Interface* `gc_port_if` – zu implementieren von jeder Klasse, die Transaktionen entgegen nehmen soll – enthält eine virtuelle Funktion, deren Implementierung eine Transaktion entgegen nimmt (`transport(ControlTransactionHandle& tr)`).

Der *Core* (Klasse `GC_Core`) bekommt über das von ihm implementierte Port-Interface Transaktionen, die er verzögerungsfrei entsprechend ihrer enthaltenen Adressierung an eines der angeschlossenen Service-Plug-ins oder an eine der User-APIs weiterleitet. Der Core ist also ein Router. Der Adressierungsmechanismus wird in Abschnitt 4.2.1 beschrieben. Der Core ist als Singleton ausgeführt, das heißt es gibt in der Simulation nur eine Instanz. Die statische Funktion `get_instance` liefert den Core zurück und erzeugt ihn falls notwendig. Das Klassendiagramm in Abbildung 4.5 ist vereinfacht: Die Klasse `GC_Core` ist in der Implementierung (vgl. Projektquellen [Schr11]^i) tatsächlich ein Synonym (`typedef`) für die Template-Klasse `GC_Core_T` mit dem Template-Parameter `gc_port`[6]. Die Funktionen `bind` und `unbind` werden von den Ports beim Verbinden bzw. beim Aufräumen aufgerufen. Die Bedeutung des Attributs `m_pLogger` zum Loggen von Transaktionen wird in Abschnitt 4.4 beschrieben. Das Attribut `m_callbackDisp` und die dazugehörige Zugriffsfunktion `get_callbackDispatcher` dienen der in Anforderung KA15 geforderten Unauffälligkeit für SystemC-Analysen, vgl. Abschnitt 4.2.4.

Ein *Service-Plug-in* ist die einzige Implementierung eines Services. Es soll die zentral benötigten Dienste des Services zur Verfügung stellen, sodass die User-APIs nur noch den Zugriff ermöglichen müssen. Es besitzt einen Port, der den Service beim Core registriert. Es implementiert das Port-Interface, um vom Core die Transaktionen der User-APIs empfangen

[4]Unter dem Begriff Convenience wird eine Erleichterung für den Benutzer (z.B. in Form von zusätzlichen Funktionen oder Klassen) verstanden, die nicht notwendig wäre, um die geforderte Funktionalität zu erfüllen, dem Benutzer aber komplexe, umfangreiche, fehleranfällige oder sich wiederholende Arbeit abnimmt.

[5]Factory: Eine Factory ist ein Konstrukt, das auf einen bestimmten Typ (hier Transaktionen) festgelegt wird und mehrere Objekte diesen Typs in einem Pool erstellt, diese dem Benutzer bei Bedarf sehr speichereffizient zur Verfügung stellt und nach deren Rückgabe in den Pool immer wiederverwendet.

[6]Dieses ist ein in der Implementierung des Öfteren verwendeter Kniff, um eine zyklische Include-Abhängigkeit zwischen den Header-Dateien zu vermeiden. Aus Gründen der Übersichtlichkeit und Verständlichkeit wird im weiteren Verlauf des Dokuments darauf verzichtet, diesen implementiertungstechnischen Umstand einer Template-Klasse, die nur für einen Datentyp sinnvoll ist und mit einem entsprechenden Synonym versehen ist, zu nennen. Der Klassenname einer solchen Klasse wird im Dokument mit dem Namen des entsprechenden Synonyms ersetzt.

zu können. Um vom Plug-in initiierte Transaktionen an bestimmte User-APIs zu senden, kann der Port genutzt werden. Das Service-Plug-in soll vom `command_if`-Interface ableiten, um das in Abschnitt 4.4 beschriebene Debuggen zu ermöglichen. Jedes Plug-in repräsentiert genau einen Service, der dem Port mitgeteilt und von dort an den Core weitergegeben wird.

Das Beispiel-Plug-in `Ex_Service_Plugin` in Abbildung 4.5 verwendet den Service-Erweiterungsmechanismus und leitet zu diesem Zweck von einer Service-ID ab. Diese ID wird dem Port übergeben. Details zum Erweiterungsmechanismus werden in Abschnitt 4.3 gegeben.

Eine *User-API* bietet dem Endnutzer eine Schnittstelle zu einem bestimmten Service. Zu einem Service kann es verschiedene User-APIs geben, wenn verschiedene Schnittstellen oder Adapter gewünscht sind. Die User-API besitzt einen Port und verwendet diesen, um Transaktionen über den Core an das zugehörige Service-Plug-in zu senden und implementiert das Port-Interface, um Transaktionen, z.B. vom Service-Plug-in, empfangen zu können. Auch User-APIs sollen das `command_if`-Interface implementieren. Es bleibt dem Entwickler des Services überlassen, ob eine User-API systemweit als Singleton angelegt werden soll – mit einer statischen Zugriffsfunktion –, von jedem Modell eine eigene Instanz erzeugt oder ein Zwischenweg gewählt werden soll. Für Debug-Zwecke kann es sinnvoll sein, für jedes Modell eine eigene Instanz anzubieten, um die Herkunft von Transaktionen besser bestimmen zu können.

4.2.1. Adressierung

In diesem Abschnitt wird der Adressierungsmechanismus von Transaktionen beschrieben. Es gibt in einer Transaktion drei Adresstypen:

1. *Target-Adresse*: Die Target-Adresse (Typ `cport_address_type`) ist ein `typedef` auf einen `gc_port_if`-Pointer. Mit dieser Adresse kann eine Transaktion von einem Service-Plug-in oder einer User-API direkt zu einer User-API gesendet werden. Die Adresse darf Null sein, in dem Fall wird alternativ die Service-Adresse verwendet. Die Target-Adresse hat Priorität vor der Service-Adresse.

2. *Service-Adresse*: Die Service-Adresse (Typ `ControlService`) zeigt auf einen Service und wird von der Factory des Ports automatisch eingetragen. Wird keine Target-Adresse gesetzt, wird die Transaktion vom Core an dieses für den Service zuständige Service-Plug-in geroutet. Die Service-Adresse kann sowohl Standard-Services als auch erweiterte Services adressieren, da sie ein `unsigned int` ist, dessen untere Werte durch eine Aufzählung gegeben sind, siehe Abschnitt 4.2.3.

3. *Sender-Adresse*: Die Sender-Adresse ist der Absender der Transaktion, d.h. diejenige Instanz, die das Port-Interface implementiert, typischerweise also der Besitzer des Ports. Die Adresse ist wie die Target-Adresse ein `gc_port_if`-Pointer und kann folglich vom Empfänger der Transaktion verwendet werden, um im weiteren Verlauf Transaktionen an den Absender zurückzuschicken.

Eine User-API, die eine Transaktion aus der Factory erhält, muss keine Target-Adresse setzen, da die Transaktion automatisch mit der Service-Adresse, zu der die User-API und damit der Port gehören, versehen ist. Auch die Sender-Adresse wird automatisch vom Port eingetragen.

Die Adresstypen sind Attribute in der Transaktion, deren Details im folgenden Abschnitt 4.2.2 beschrieben werden.

Es muss beachtet werden, dass während der Middleware-Kommunikation kein SystemC-`wait` aufgerufen werden darf, da Transaktionen nicht verzögert werden und die Simulation nicht durch die Middleware beeinflusst werden soll.

4.2.2. Transaktionen

Transaktionen sind die Datenstrukturen der GreenControl-Modell-Middleware, die über die statische API der Middleware übertragen werden können und trotzdem dynamisch erweiterbar sind. Transaktionen übertragen beispielsweise die Parameter der User-API-Aufrufe an das Service-Plug-in und transportieren die Rückgabewerte.

Eine Transaktion besteht aus Attributen, die über `set-` und `get`-Funktionen gesetzt und gelesen werden können. Zusätzlich bietet sie einen Erweiterungsmechanismus, der in Abschnitt 4.2.3 näher erläutert wird.

Transaktionen sollen von den User-APIs und Plug-ins immer aus der Factory des Ports entnommen werden, der das Memory-Management vornimmt (siehe unten). Die speziellen Fähigkeiten einer Transaktion zum Loggen werden im Abschnitt 4.4 eingeführt.

Aufbau aus Service-Sicht

Tabelle 4.6 listet die Attribute der Transaktionen auf, die aus Service-Sicht interessant sind. Die Attribute werden in Kategorien eingeteilt:

Die Attribute der Kategorie Routing wurden bereits in Abschnitt 4.2.1 beschrieben. Sie sind als übergeordnet markiert, da sie für die Funktion der Middleware selbst notwendig sind. Die weiteren Atribute sind servicespezifisch, was bedeutet, dass ihre Verwendung und Bedeutung vom Service frei wählbar sind – auch wenn einige Attribute mit einer bestimmten Semantik empfohlen werden:

Das Attribut `mError` (Kategorie Fehler) ist für die Rückgabe oder das Senden von Fehlern vorgesehen. Das Attribut ist vom Typ `unsigned int`, wobei alle Zahlen $\neq 0$ Fehler darstellen sollen, deren Kodierung vom Service festgelegt werden soll. Das Attribut `mCmd` (Kategorie Kommando) ist das Kommando, das auf Empfängerseite ausgeführt werden soll. Die Semantik muss ebenfalls vom Service festgelegt werden. Das Attribut `mMetaData`, das vom Typ `string` und der Kategorie Debug-Daten ist, ist für das Anhängen von (z.B. menschenlesbaren) Debug-Informationen an die Transaktion gedacht. Das können Informationen für den Empfänger oder für eine Kommunikationsanalyse nach Abschnitt 4.4 sein. Dieses Attribut soll nicht verwendet werden, um eine Funktionalität des Services bereitzustellen,

Attribut	Kategorie	Beschreibung
mService	Routing (Ü)	Service-Adresse (Adresse eines Service-Plug-ins)
mTarget	Routing (Ü)	Target-Adresse (Adresse einer User-API)
mID	Routing (Ü)	Sender-Adresse (Adresse des Senders)
mError	Fehler (S)	Fehlercode (0: kein Fehler)
mCmd	Kommando (S)	servicespezifisches Kommando
Erweiterungen	Daten (S)	Übertragung beliebiger Daten (vgl. Abschnitt 4.2.3)
mMetaData	Debug-Daten (S)	Übertragung beliebiger `string`-Zeichenketten für Debug-Informationen

Tabelle 4.6.: Transaktion Standard-Attribute (Ü: übergeordnet; S: servicespezifisch)

damit der Inhalt zu Debug-Zwecken geändert werden kann. Unter dem Punkt Erweiterungen sind sämtliche Daten zusammengefasst, die der Service nach Bedarf mit Hilfe von den im Abschnitt 4.2.3 detailliert vorgestellten Erweiterungen in die Transaktion einfügen kann.

Abbildung 4.7 zeigt das Klassendiagramm einer Transaktion. Die Attributdefinition des Diagramms ist unterteilt in die oben beschriebenen Standard-Attribute für die Informationsübertragung und weitere Attribute. Die Operationen sind untergliedert in Zugriffsfunktionen auf die Standard-Attribute und deren Reset, Funktionen, die das Debuggen unterstützen, Funktionen für den Log-Mechanismus (vgl. Abschnitt 4.4) und Funktionen für den Erweiterungsmechanismus (vgl. Abschnitt 4.2.3).

Memory-Management

Der Port besitzt eine Transaktions-Factory mit einem Pool, die beim Aufruf der Funktion `createTransaction` einen `shared_ptr`[7] auf eine Transaktion zurückgibt (der Rückgabetyp ist `ControlTransactionHandle`).

Wenn eine Transaktion von einer User-API oder einem Service-Plug-in zum Senden benötigt wird, ruft sie diese Funktion auf ihrem Port auf und speichert die Rückgabe in einem lokalen Shared-Pointer. Sie kann diese Transaktion so lange verwenden, wie sie den Shared-Pointer speichert. Das ist typischerweise nur lokal innerhalb der User-API- oder Plug-in-Funktion der Fall, darf aber auch länger sein. Das Freigeben ist nicht teuer, da von der Factory nur ein Reset auf dem Objekt aufgerufen wird und sie zurück in den Pool gegeben wird. Dieses Verfahren sichert eine effiziente Nutzung von Transaktions-Objekten.

Obwohl das Kopieren von Transaktionen möglich ist, wird das nicht empfohlen, da von den Pointer-Attributen und Erweiterungen nur die Adressen kopiert werden, also keine Deep-Copy stattfindet.

Für das Memory-Management der Erweiterungen ist Abschnitt 4.2.3 zu beachten.

[7] Ein Shared-Pointer ist ein spezieller Pointer auf einen Datentyp, der das Objekt erst freigibt, wenn es keine Referenz mehr auf das Objekt gibt.

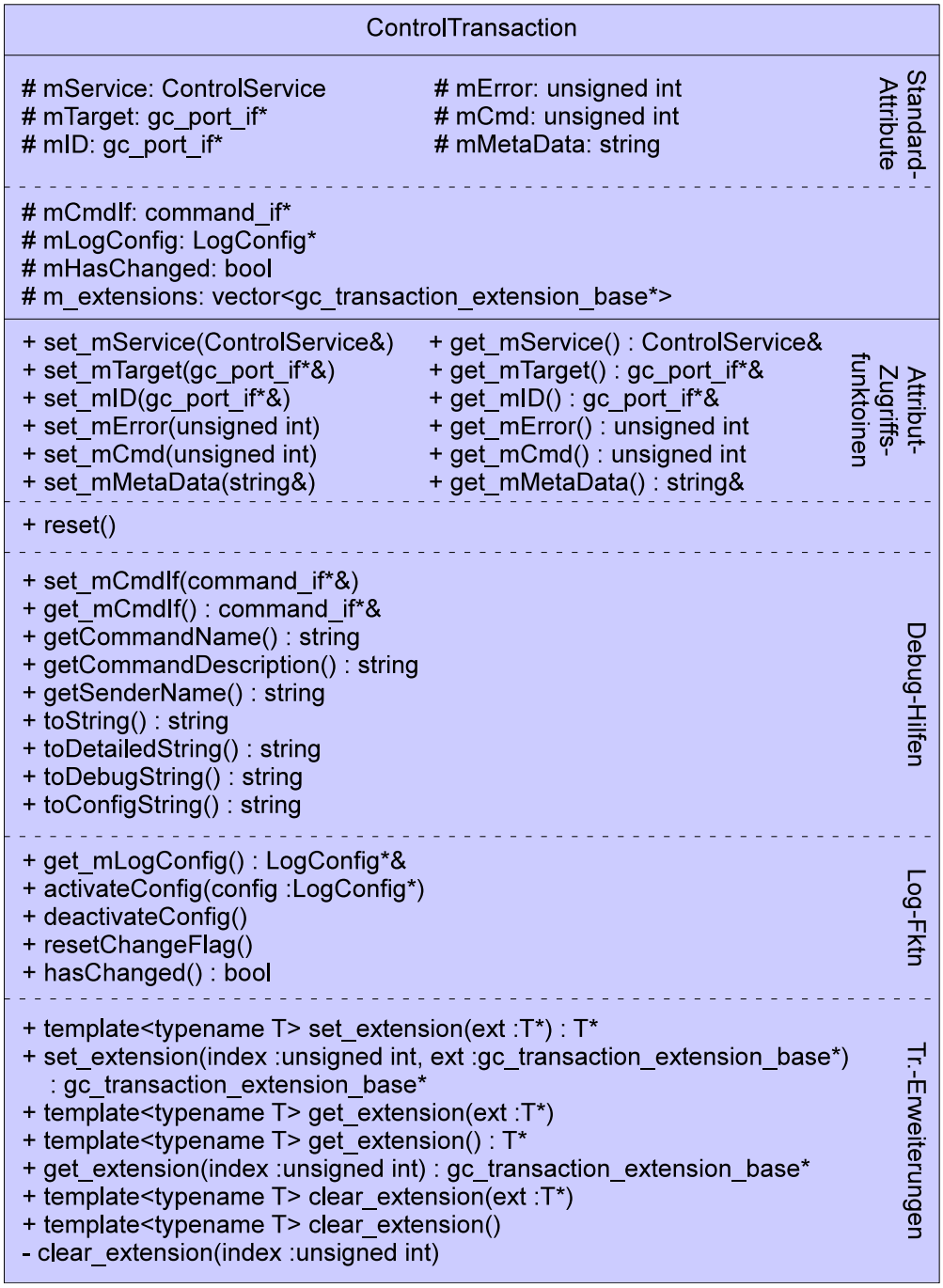


Abbildung 4.7.: Transaktionen (Klassendiagramm)

4.2.3. Erweiterte Transaktionen

Für die Kommunikation zwischen User-APIs und Service-Plug-ins ist die Übertagung von Informationen über Transaktionen notwendig (Kategorie Daten aus Tabelle 4.6 auf Seite 78). Dafür kann der Service beliebige Erweiterungen definieren, die an eine Transaktion angefügt werden.

Die folgenden Alternativen stehen für die jeweiligen Anwendungsfälle für die Übertragung von Informationen in den Attributen und Erweiterungen der Transaktionen zur Verfügung:

1. *Vorhandene Attribute verwenden:* Solche Attribute sind für wiederkehrende Aufgaben vorgesehen:
   - Die Identifizierung des auszuführenden Befehls und damit die Existenz und die Bedeutung der Erweiterungen soll über das Attribut `mCmd` erfolgen. Die Kodierung wird vom Service festgelegt. Für einen bestehenden Service kann ein neues Kommando eingeführt werden, ein neuer Service soll eigene Kommandos definieren.
   - Die Fehlerbehandlung soll über das Fehler-Attribut `mError` signalisiert werden, der Wert 0 soll für Fehlerfreiheit stehen.[8].
   - Das Attribut `mMetaData` kann für die Übertragung von Debug-Informationen verwendet werden. Der Service soll seine Funktion nicht auf dieses Attribut stützen, damit es noch bedarfsbezogen geändert werden kann.
2. *Den Erweiterungsmechanismus nutzen:* Die Übertragung von Nutzdaten soll über den im Folgenden beschrieben Erweiterungsmechanismus erfolgen. Er erlaubt die Definition eigener Erweiterungen, von denen je eine pro Typ an jede Transaktion gehängt werden kann.

Der Erweiterungsmechanismus für GreenControl-Transaktionen ist an den Erweiterungsmechanismus für TLM-2.0-Transaktionen angelehnt (vgl. [OSCI09]). Da in der Modell-Middleware keine (SystemC-)Zeit vergehen darf, ist das Memory-Management jedoch sehr viel einfacher, wodurch sich auch die Erweiterungsklassen im Vergleich zu TLM-2.0 stark vereinfachen.

Transaktionserweiterung

Das Klassendiagramm in Abbildung 4.8 zeigt den Ableitungsbaum einer Beispiel-Transaktionserweiterung `Ex_Transaction_Extension`.

Eine Transaktionserweiterung kann beliebige Datenstrukturen beinhalten; im Beispiel ist es das Attribut `m_expl_extension_data`, das vom imaginären Typ `anything` ist. Die Erweiterung muss von der Klasse `gc_transaction_extension` ableiten und diese an den eigenen

[8] Für die Fehlersignalisierung können neben dem Fehler-Attribut zusätzliche Erweiterungen genutzt werden.

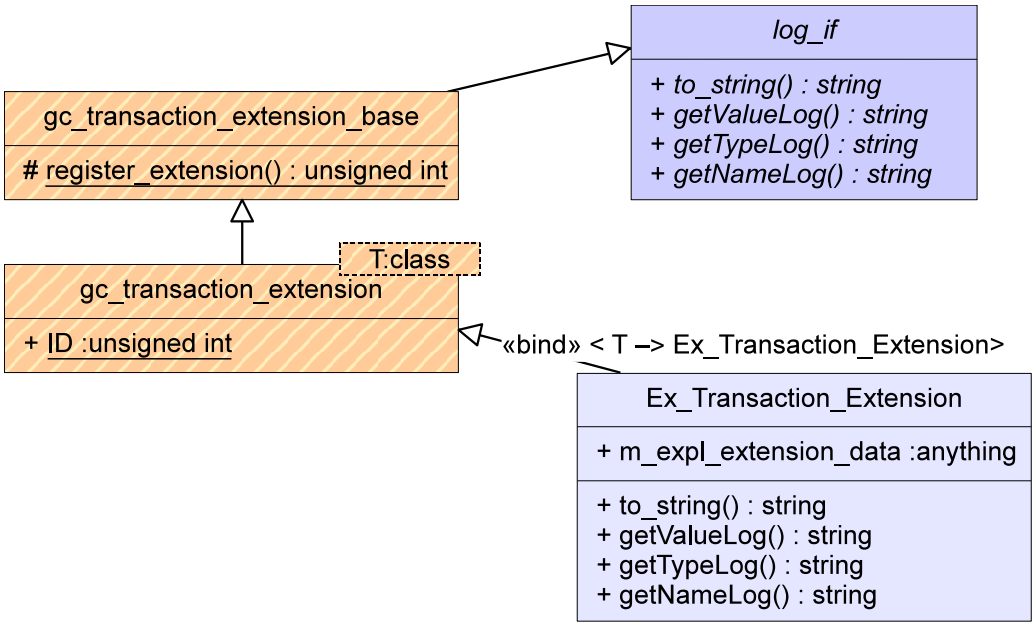


Abbildung 4.8.: Transaktionserweiterung (Klassendiagramm)

Typ binden. Das bewirkt, dass die Erweiterung während der statischen Konstruktion einen simulationsweit eindeutigen Index `ID` bekommt. Dieser Index ist auch die Position der Erweiterung im Erweiterungsarray `m_extensions` einer `ControlTransaction`. Folglich kann es auch nur eine Erweiterung jeden Typs in einer Transaktion geben.

Für die simulationsweite Eindeutigkeit des Index sorgt die statische Registrierungsfunktion `register_extension` der Basisklasse `gc_transaction_extension_base` und die (klassenlose) Funktion `max_num_gc_transaction_extensions`. Letztere Funktion zählt das globale Maximum der Menge von Transaktionserweiterungen. Ein Inkrementieren findet jedes Mal statt, wenn eine neue Erweiterungklasse definiert wird – was während der statischen Initialisierung geschieht. Das zurückgegebene globale Maximum wird – nach der statischen Initialisierung – im Konstruktor der Transaktion verwendet, um die Größe des Erweiterungsarrays festzulegen.

Die Funktionen des Interfaces `log_if` sind rein virtuell, sodass sie von jeder Transaktionserweiterung implementiert werden müssen, damit die Erweiterung von der in Abschnit 4.4 beschriebenen internen Kommunikationsanalyse untersucht werden kann.

Anwendung von Transaktionserweiterungen

Eine Transaktion kann beliebig viele[9] *verschiedene* Transaktionserweiterungen aufnehmen, allerdings von jeder Transaktionserweiterung nur eine.

Die Erweiterungen von Transaktionen erfüllen die Anforderung AA5, die interne API ohne Neukompilieren flexibel zu gestalten, da sie ohne Änderung des Middleware-Interfaces übertragen werden können.

[9]Die Höchstgrenze ist theoretisch durch die Kapazität eines `unsigned int` begrenzt.

Der Empfänger der erweiterten Transaktion sollte sinnvollerweise mit der Erweiterung umgehen können, das muss aber nicht der Fall sein; beispielsweise könnte der Sender verschiedene Empfänger unterstützen, also abwärtskompatibel sein. Ein Service kann um neue Merkmale erweitert werden, indem eine neue User-API neue Transaktionserweiterungen verwendet. Diese neuen Erweiterungen muss das Service-Plug-in verstehen, allerdings können bestehende User-APIs inklusive deren Implementierungen erhalten bleiben (sie müssen nicht einmal neu kompiliert werden). In diesem Beispiel kann das Service-Plug-in abwärtskompatibel bleiben, indem es mit Transaktionen mit und ohne dieser neuen Erweiterung umgehen kann.

Das Hinzufügen und Abfragen von Erweiterungen wird über Zugriffsfunktionen[10] realisiert (vgl. Abbildung 4.7, Unterteilung Tr.-Erweiterungen). Es gibt drei verschiedene Arten von Funktionen, deren Vertreter die betreffende Erweiterung jeweils anhand des Typs oder der ID identifizieren:

1. *Hinzufügen einer Erweiterung:* Die Funktionen `set_extension` fügen die übergebene Erweiterung der Transaktion hinzu. Ein eventuell bereits vorhandener Eintrag wird überschrieben, ohne das Objekt zu löschen. Die überschriebene Erweiterung (oder 0, wenn keine überschrieben wurde) wird zurückgegeben.

2. *Abfragen einer Erweiterung:* Die Funktionen `get_extension` liefern die entsprechende Erweiterung, wenn sie in der Transaktion vorhanden ist, ansonsten 0.

3. *Entfernen einer Erweiterung:* Die Funktionen `clear_extension` entfernen den Eintrag der Erweiterung aus der Transaktion und ersetzen ihn durch 0. Das Erweiterungsobjekt wird nicht gelöscht!

Das Entfernen und gleichzeitige Löschen einer Transaktionserweiterung (vgl. Release-Funktionen in TLM-2.0) ist nicht vorgesehen, da der Sender das Memory-Management sehr einfach selbst übernehmen kann, da GreenControl-Transaktionen nicht über die Dauer des Transportaufrufs hinaus verwendet werden. Der Sender kann beispielsweise einen eigenen Pool (oder einzelne Objekte) verwalten und diese den Transaktionen hinzufügen.

Der Reset einer Transaktion bewirkt das Entfernen aller Erweiterungen (wiederum ohne die Objekte zu löschen). Eine Transaktion aus dem Pool des Ports enthält also keine Erweiterungen und Erweiterungen können nach der Rückkehr der Transaktion in den Pool vom Sender wiederverwendet oder gelöscht werden.

4.2.4. Callback-Dispatcher

Das in diesem Abschnitt beschriebene Konstrukt dient der Anforderung KA15, nach der die Middleware in möglichen SystemC-Analysen möglichst unauffällig sein soll.

[10] Die Funktionen zum Hinzufügen, Zugreifen und Entfernen von Erweiterungen sind denen von TLM-2.0-Erweiterungen ähnlich.

GreenControl bietet den Services zentralisiert die Möglichkeit, SystemC-Elaborations- und Simulations-Callbacks zu erhalten, ohne ein `sc_module` sein zu müssen. Das minimiert die Sichtbarkeit der Middleware und deren Services, ohne ihnen die wichtige Möglichkeit zu nehmen, SystemC-Callbacks empfangen zu können. Zwar soll die Middleware unauffällig bezüglich SystemC-Analysen sein, allerdings muss sie mindestens ein `sc_module` besitzen, das diese Callbacks vom Kernel empfängt und weiter verteilt, damit die Modell-Middleware unabhängig vom verwendeten SystemC-Kernel bleibt[11]. Dieses Modul ist der **Callback-Dispatcher**.

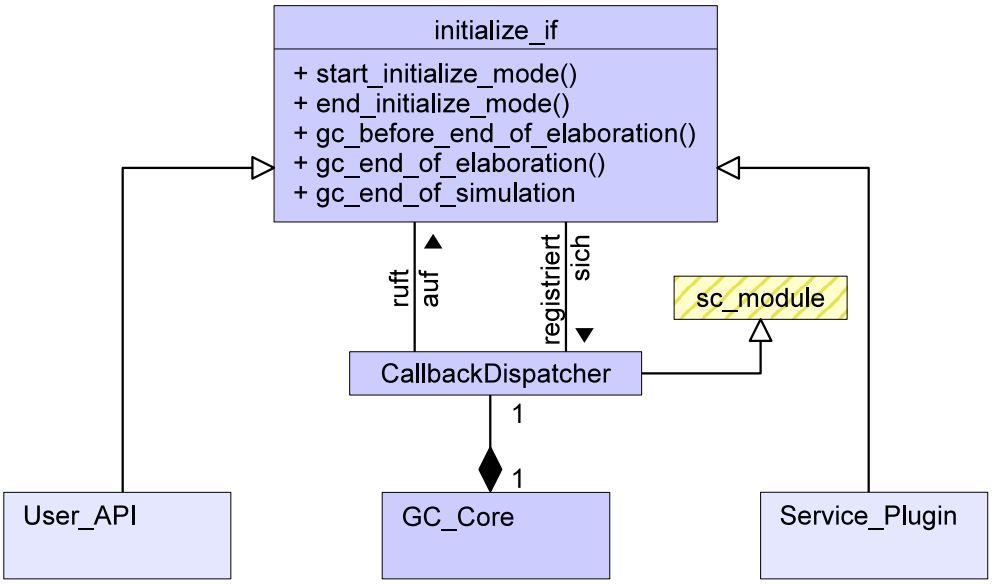


Abbildung 4.9.: Callback-Dispatcher (Klassendiagramm)

Der Callback-Dispatcher (Klasse `CallbackDispatcher`) wird vom Core erzeugt und in seinem Attribut `m_callbackDisp` gespeichert (vgl. Klassendiagramm in Abbildung 4.5 auf Seite 74). Der Callback-Dispatcher existiert folglich wie der Core nur einmal in der Simulation. Abbildung 4.9 zeigt die Zusammenhänge mit dem Core und den Services, deren Plug-ins und User-APIs den Callback-Dispatcher beliebig verwenden dürfen. Das Klassendiagramm zeigt, dass es mit dem Callback-Dispatcher nur ein Objekt gibt, das ein SystemC-Modul (in der Abbildung leicht schraffiert) ist.

Um die Verwendung für die Services so einfach wie möglich zu halten, können die Serviceklassen vom Interface `initialize_if` ableiten. Der Konstruktor dieses Interfaces registriert sich automatisch beim Callback-Dispatcher und somit werden sofort alle Callbacks weitergeleitet, vorausgesetzt die entsprechenden virtuellen Funktionen wurden in der Serviceklasse auch definiert. Die folgenden Elaborations- und Simulations-Callbacks werden weitergegeben:

[11] Mit einer Modifikation des jeweils verwendeten SystemC-Kernels ließe sich der Callback-Dispatcher leicht umbauen, sodass er kein SystemC-Modul mehr ist. Das wäre optimal bezüglich der Unauffälligkeit, wäre dann allerdings nicht mehr kernelunabhängig und standardkonform

- Der SystemC-Callback `before_end_of_elaboration` wird an die Interface-Funktion `gc_before_end_of_elaboration` weitergegeben.
- Der SystemC-Callback `end_of_elaboration` wird an die Interface-Funktion `gc_end_of_elaboration` weitergegeben.
- Der SystemC-Callback `start_of_simulation` wird an die Interface-Funktion `gc_start_of_simulation` weitergegeben.
- Der SystemC-Callback `end_of_simulation` wird an die Interface-Funktion `gc_end_of_simulation` weitergegeben.

Zusätzlich unterteilt der Callback-Dispatcher den `start_of_simulation`-Callback in zwei Callbacks, die innerhalb eines Services verwendet werden können. Dieser zusätzliche Modus wird **Initialize-Mode** genannt: Wenn der Callback vom SystemC-Kernel beim Callback-Dispatcher aufgerufen wird, wird zuerst auf allen registrierten Objekten die den Beginn anzeigende Funktion `start_initialize_mode` aufgerufen. Dann erst wird der Callback an die Funktionen `gc_start_of_simulation` weitergegeben. Anschließend wird auf allen Objekten die Funktion `end_initialize_mode` aufgerufen. Der erste Aufruf markiert den Eintritt in den Initialize-Mode, der letzte das Ende. Dieser Modus kann von den Services nach Bedarf benutzt werden. Den zusätzlichen Nutzen gegenüber den weitergegebenen SystemC-Callbacks liefert die zusätzliche Signalisierung des Endes.

4.3. Services

Dieser Abschnitt beschreibt zusammenfassend die Eigenschaften von GreenControl-Services und stellt das Erweiterungskonzept vor.

Die Service-Plug-ins und User-APIs verwenden einen gemeinsamen Satz Kommandos, die über Transaktionen zusammen mit sämtlichen zu übertragenden Daten in beide Richtungen übertragen werden (vgl. Abschnitt 4.2.2). Das Routing zum Plug-in geschieht automatisch, alternativ können User-APIs explizit adressiert werden (vgl. Abschnitt 4.2.1). Die Semantik der Kommandos und Daten in den Transaktionen muss der Service festlegen. Transaktionserweiterungen können transparent hinzugefügt werden (vgl. Abschnitt 4.2.3). Das Senden von Transaktionen wird über instanziierte Ports durchgeführt, das Empfangen wird durch das Implementieren der `transport`-Funktion im `gc_port_if` möglich.

Systemweit werden alle Services mit einer **GreenControl-Service-ID** identifiziert. Der Datentyp `ControlService` ist ein Synonym für einen `unsigned int`. Die Middleware unterscheidet zwischen fest eingebundenen und erweiterten Services: Fest eingebundene Services werden in der Aufzählung (`enum`) `ControlService_enum` gelistet. Beispielsweise sind das der Konfigurations-Service (Enumerator `CONFIG_SERVICE`) und der Analyse-Service (Enumerator `AV_SERVICE`) (siehe Kapitel 5 und Anhang A). Die den Enumeratoren fest zugeordneten Werte sind gleichzeitig deren Service-IDs. Fest eingebundene Services haben die Eigenschaft,

dass deren Aufzählung nicht ohne Neukompilieren der Middleware verändert werden kann. Deswegen gibt es den Erweiterungsmechanismus für Services[12]:

Neue Services

Die Modell-Middleware bietet einen Erweiterungsmechanismus für neue Services.

Das Klassendiagramm in Abbildung 4.5 auf Seite 74 zeigt bereits die notwendige Ableitungshierarchie (schraffiert) aus Benutzersicht: Für einen neuen Service muss zunächst eine Erweiterungsklasse definiert werden, in Abbildung 4.5 ist das die Klasse `Ex_Service_ID`. Diese Erweiterungsklasse leitet von der Klasse `gc_service_ID_extension` ab und bindet diese an den eigenen Typ. Alle Besitzer eines Ports müssen von dieser Serviceerweiterungsklasse ableiten, um sich und den Port gegenüber dem Core identifizieren zu können.

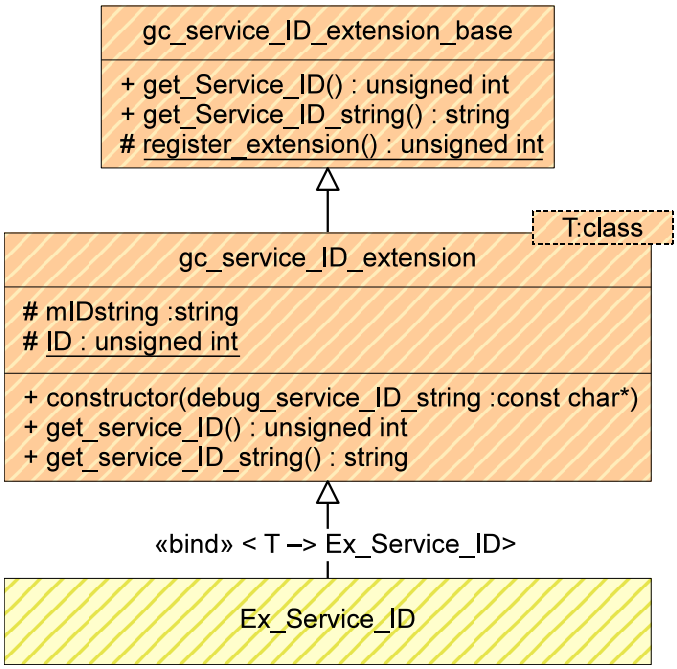


Abbildung 4.10.: Serviceerweiterung (Klassendiagramm)

Die interne Klassenstruktur der Serviceerweiterung ist in Abbildung 4.10 abgebildet. Sie ähnelt der der Transaktionserweiterung und verwendet ebenfalls das TLM-2.0-Erweiterungskonzept. Serviceerweiterungen verwenden das gleiche Prinzip des simulationsweit eindeutigen Index, der während der statischen Konstruktion berechnet wird. Die entsprechende Funktion dafür ist `max_num_gc_service_ID_extensions`.

Die Erweiterungsbasisklasse `gc_service_ID_extension_base` bietet eine virtuelle Funktion an[13], die die Service-ID der abgeleiteten Serviceerweiterung liefert. Das ermöglicht dem Port und dem Core die Abfrage der ID ohne die Klasse zu kennen.

[12] Es gibt keine Konfigurationsmechanismus-Anforderung für dieses Erweiterungskonzept, da es andere als den Konfigurations-Service ermöglichen soll.
[13] Eine weitere Funktion liefert die zur Service-ID zugeordnete Zeichenkette für Debug-Zwecke.

Die Erweiterungsklasse `gc_service_ID_extension` speichert die Service-ID (Attribut `ID`) und die zugehörige Debug-Zeichenkette, die von der Serviceerweiterung festgelegt wird. Zugriffsfunktionen geben Zugriff darauf. Der Konstruktor erwartet eine Zeichenkette, die als Service-Name für Debug-Zwecke verstanden werden kann.

Ein Beispiel für einen Service, der den Erweiterungsmechanismus nutzt, ist im externen Listing B.1 im Anhang aufgeführt.

4.4. Interne Analyse und Debugfähigkeiten

In diesem Abschnitt wird zunächst eine Übersicht über verschiedene Funktionen in der Modell-Middleware beschrieben, mit denen menschenlesbare Zeichenketten für interne Zusammenhänge erlangt werden können. Im Anschluss wird die Kommunikationsanalyse mit einem Logger vorgestellt.

- *Service-ID-Zeichenkette:* Die globale Funktion

  ```
  const std::string getControlServiceString(ControlService cs)
  ```

  liefert eine Zeichenkette für eine übergebene Service-ID. Sie verwendet intern die bereits für Serviceerweiterungen vorgestellte Funktion `get_Service_ID_string` der Klasse `gc_service_ID_extension`.

- *Kommando-Zeichenkette:* Die API- und Plug-in-Schnittstelle `command_if` (siehe Abbildung 4.5 auf Seite 74) soll von den User-APIs und Service-Plug-ins implementiert werden. Sie bietet Zugriff auf die Namen der von dem entsprechenden Service verwendeten Kommandos (`getCommandName`) und deren Beschreibung (`getCommandDescription`). Zudem ist die Abfrage des Sender-/Empfängernamens möglich (`getName`). Das Kommando-Interface kann vor allem bei der Darstellung des Inhalts und Routinginformationen von Transaktionen helfen.

- *Transaktionen:* Transaktionen bieten einfache Funktionen, die den Inhalt der Transaktionen als Komplettzeichenkette liefern: `toString` und `toDetailedString` (mit Logger-Verwendung: `toDebugString`, `toConfigString`).

- *Inhalt von Transaktionen:* Um weiteren Inhalt von Transaktionen (d.h. Erweiterungen und das Log-Pointer-Attribut) darstellen und untersuchen zu können, gibt es die `log_if`-Schnittstelle (Details siehe folgenden Unterabschnitt). Sie liefert zu diesem Zweck verschiedene Arten von Zeichenketten:

  - `getValueLog`:
    Diese Funktion soll die Daten in möglichst lesbarer Form zurückgeben.
  - `getTypeLog`:
    Diese Funktion soll den Datentyp der Daten menschenlesbar zurückgeben.

- `getNameLog`:
  Diese Funktion soll – falls es sich um ein SystemC-Objekt handelt – den SystemC-Namen zurückgeben, andernfalls einen anderen Namen.

- `toString`:
  Diese Funktion soll die anderen Zeichenketten in geeigneter Weise kombinieren.

Kommunikationsanalyse mit GreenControl-Logger

Die Middleware bietet sich als zentraler Analysepunkt für Service-Aktionen an. Eine Analyse der Transaktionen, die zwischen Service-Plug-ins und User-APIs ausgetauscht werden, kann wertvolle Informationen für das Debuggen von Services während deren Entwicklung liefern. Je nach Service könnte eine solche Kommunikationsanalyse auch beim Debuggen von Modellen helfen. In diesem Abschnitt wird der für diese Aufgabe vorgesehene generische **GreenControl-Logger** beschrieben, der von Michael Rütz in seiner Diplomarbeit entwickelt wurde [Ruet10]. Der Logger kann sämtliche zwischen User-APIs und Service-Plug-ins ausgetauschten Transaktionen beobachten, filtern, aufzeichnen und für die Analyse aufbereiten. Der Logger ist anpassungsfähig und kann generisch für unbekannte Services eingesetzt werden. Dazu können Filter und Ausgabemodule umfassend konfiguriert und kombiniert werden.

Der Logger bietet den *Log-Service* mit der Service-ID `LOG_SERVICE` an und wird wie ein Service-Plug-in automatisch an den Core angeschlossen. Dabei wird er vom Core anhand der Service-ID erkannt und bekommt eine Sonderstellung: Er wird vom Core im Klassenmember `m_pLogger` gespeichert, was automatisch den Log-Vorgang aktiviert. Von dem Zeitpunkt an bekommt der Logger sämtliche Transaktionen, die vom Core transportiert werden, zweimal übergeben: einmal (an die Funktion `log`) bevor sie an den Empfänger ausgeliefert werden und einmal (an die Funktion `logReturn`) nachdem der Aufruf von dort zurückgekehrt ist. Die zweite Übergabe dient dem Loggen von Änderungen, also Rückgabewerten. Das Sequenzdiagramm in Abbildung 4.11 illustriert den Ablauf anschaulich.

Das Klassendiagramm in Abbildung 4.12 zeigt den Aufbau: Die Klasse `GC_Logger` existiert als Singleton. Um sich automatisch mit dem Core zu verbinden, besitzt der Logger einen Port und leitet vom Port-Interface ab. Damit besitzt der Logger zunächst die gleichen Kommunikationsmerkmale wie herkömmliche Service-Plug-ins. Der Port könnte zum Senden und Empfangen von Transaktionen verwendet werden. Um jedoch die herkömmliche Service-Kommunikation nicht zu beeinträchtigen, leitet der Logger zusätzlich vom *Logger-Interface* `GC_Logger_if` ab[14]. Die dort zu implementierenden Funktionen haben die gleiche Signatur wie die herkömmliche `transport`-Funktion und erhalten zu den oben genannten Zeitpunkten die Transaktionen zum Loggen übergeben. Es werden zwei verschiedene Funktionen verwen-

[14]Die herkömmliche Service-Kommunikation wird vom Log-Service nicht genutzt, da er ja diese Art von Kommunikation untersuchen soll und sich somit absichtlich nur auf andere Wege konfigurieren und steuern lässt.

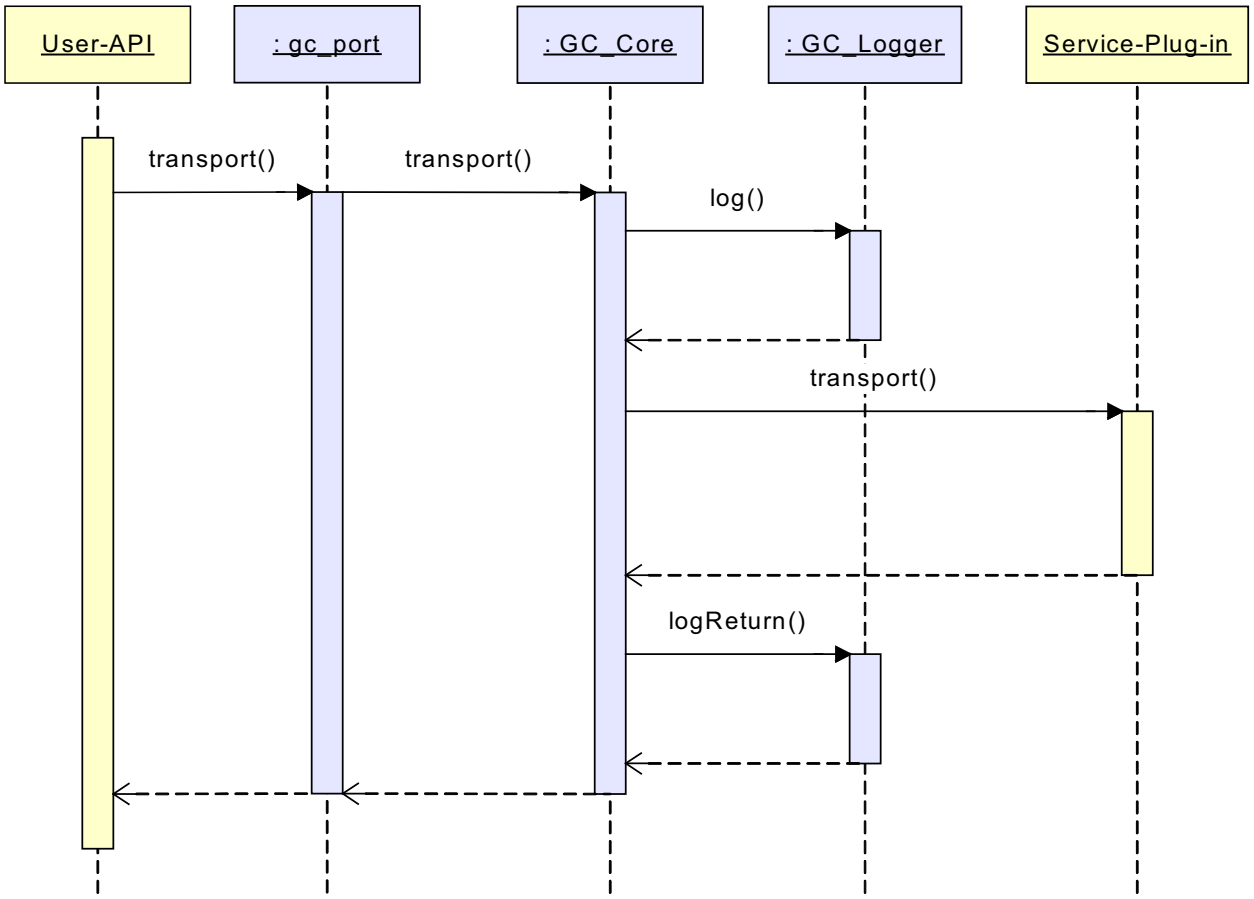


Abbildung 4.11.: Ablauf Logger (Sequenzdiagramm, nach [Ruet10])

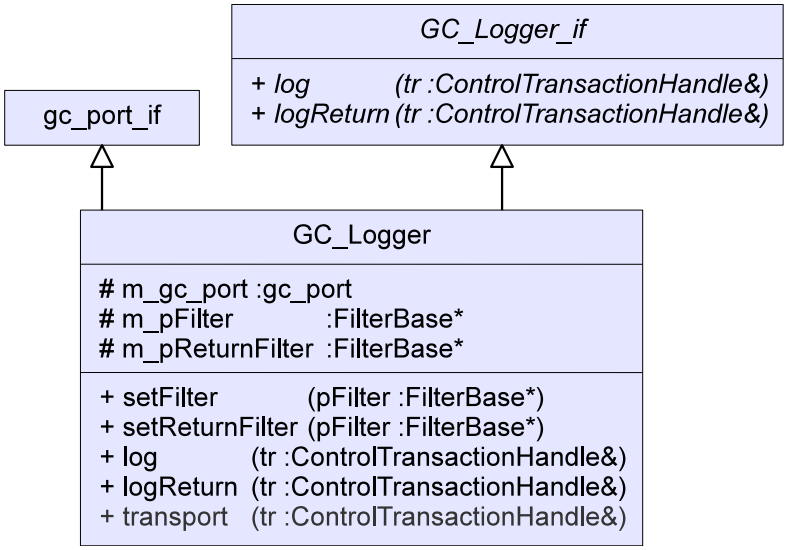


Abbildung 4.12.: GreenControl-Logger (Klassendiagramm)

det, damit keine dynamische Unterscheidung von Aufruf und Rückkehr notwendig ist, die besonders negative Auswirkungen auf die Performance hätte.

Der Logger und seine Filter sollten feststellen können, ob eine Transaktion zwischen den beiden Aufrufen der Logger-Interface-Funktionen verändert wurde und welche Attribute betroffen sind. Sonst würden potentiell zu große und unübersichtliche Datenmengen entstehen. Zu diesem Zweck bietet die Transaktion das Changed-Flag `mHasChanged` und die Funktionen `hasChanged` und `resetChangeFlag` (vgl. Abbildung 4.7 auf Seite 79). Das Changed-Flag wird mit der Funktion `resetChangeFlag` auf `false` gesetzt, sobald die Transaktion den Logger das erste Mal erreicht. Jeder schreibende Zugriff im Empfänger auf ein beliebiges Standard-Attribut bewirkt automatisch ein Setzen des Flags auf `true`. Der Logger kann den Status auf dem Rückgabepfad mit Hilfe von `hasChanged` abfragen[15]. Da die Verwaltung des Changed-Flags mit Performance-Einbußen verbunden ist, wird dieses Merkmal mit Hilfe des Makros `GCLOG_RETURN` nur aktiviert, wenn der Benutzer es definiert.

Die aktuelle Implementierung erlaubt aktuell nur das Aufzeichnen der Standard-Attribute und aller in den Konfigurations- und Analyse-Services verwendeten Attribute.

In der Diplomarbeit [Ruet10] werden ausführliche Details zum Konzept, der Implementierung und einem umfangreichen Filtermechanismus präsentiert, mit dem dynamisch festgelegt werden kann, welche Daten geloggt und für die Analyse exportiert werden sollen.

4.5. Verwendung

Die Modell-Middleware steht im GreenControl-Projekt OpenSource auf der GreenSocs-Webseite[16] und in der Versionsverwaltung [Schr11]^i zur Verfügung. Für ausführliche Informationen aus Benutzersicht sei auf die Dokumentation des Projekts verwiesen, beispielsweise die Doxygen-Dokumentation, das Benutzerhandbuch [ScKl11b]^i und den Technischen Report [Schr11b].

[15] Ein weiterer, hier nicht vorgestellter Mechanismus erlaubt die Identifikation der modifizierten Attribute.
[16] GreenControl-Projektwebseite `http://www.greensocs.com/projects/GreenControl`

5. Konfigurations-Service

Inhalt

Dieses Kapitel behandelt eines der Hauptergebnisse dieser Arbeit, den universellen Konfigurationsmechanismus GreenConfig. Dieser baut als Service auf der in Kapitel 4 eingeführten Modell-Middleware GreenControl auf, die die notwendige flexible interne API bereitstellt und unter Beachtung der Anforderungen aus Abschnitt 3.2 entwickelt wurde.

Die Universalität dieses Mechanismus bildet die Basis für das Ziel dieser Arbeit, der Verbesserung der Austauschbarkeit von Modellen zwischen verschiedenen Entwicklungsumgebungen: Es stehen verschiedene Adapter-User-APIs zur Verfügung, und Erweiterungsmechanismen bereiten den Konfigurationsmechanismus für neue Adapter vor.

Das Kapitel beschränkt sich auf die konzeptionell interessanten Aspekte des Konfigurationsmechanismus. Umfangreiche technische Details und ausführliche Informationen aus Benutzersicht können der Dokumentation des Projektes entnommen werden, beispielsweise dem Benutzerhandbuch [ScKl11a]^i, dem Tutorial, den Vortragsfolien, der Doxygen-HTML-Dokumentation und dem Code in der Versionsverwaltung, jeweils in [Schr11]^i und dem Technischen Report [Schr11b]. Der Konfigurations-Service steht Open Source als Header-Bibliothek[1] [Schr11]^i zur Verfügung.

[1] Aufgrund des hohen Anteils an Template-Code besteht das Projekt aus Header-Dateien, die auch die Implementierung beinhalten.

5.1. Grundlegende Struktur

In diesem Abschnitt wird die grundlegende Struktur des Konfigurationsmechanismus GreenConfig beschrieben. Eine Übersicht zeigt Abbildung 5.1 auf der gegenüberliegenden Seite. Für das Konzept interessante Details werden in den weiteren Abschnitten dieses Kapitels aufgegriffen.

Der Mechanismus baut als Service auf der Modell-Middleware GreenControl auf, da diese unter Beachtung der Anforderungen an den Konfigurationsmechanismus entworfen wurde. Der wichtigste Faktor ist die Anforderung AA5 der flexiblen internen API für die Erweiterbarkeit durch neue Merkmale (AA4).

Der **Konfigurations-Service GreenConfig** erhält seine grundlegende Struktur dadurch, dass er ein Service der Modell-Middleware ist: Die zentrale Funktionalität ist in einem Konfigurations-Service-Plug-in gekapselt, der Zugriff findet über verschiedene User-APIs statt. Die User-APIs teilen sich entsprechend Abschnitt 2.5 in Modell-APIs und Tool-APIs auf. Abbildung 5.1 auf der gegenüberliegenden Seite zeigt Beispiele für die verschiedenen Arten der User-APIs.

5.1.1. Übersicht GreenConfig-Parameter

GreenConfig stellt die **GreenConfig-Parameter** `gs_param` als native Modell-API für die Verwendung in konfigurierbaren Modellen zur Verfügung (Beispiele sind die Module IP_1 und Sub_IP in Abbildung 5.1). Diese Modell-API ist der Modell-Parameter dieses Mechanismus. Die GreenConfig-Parameter verwenden den Class-Wrapper-Ansatz, da dieser gegenüber dem Konfigurations-Interface-Ansatz mächtiger ist (vgl. Abschnitte 3.1.2 und 5.9.1).

GreenConfig-Parameter sind Datentypen, die im Modell normale C++- und SystemC-Datentypen transparent ersetzen. Parameterwerte lassen sich zu Zwecken der Konfiguration schreiben. Da auch lesende, d.h. wertneutrale Zugriffe möglich sind, können sie auch für die Modell-Untersuchung verwendet werden (vgl. Abschnitt 5.10). Die Parameter können vom Benutzer nach Bedarf instanziiert werden und melden sich automatisch bei der Modell-Middleware und bei der zentralen Parameterdatenbank an. Zum Anmelden beim Service-Plug-in verwenden die Parameter die ihrem besitzenden Modul zugeordnete Tool-API (eine Beschreibung aus Sicht der Tool-API wird in Abschnitt 5.1.3 gegeben). Sie bieten zudem einen umfangreichen Callback-Mechanismus, der registrierte Beobachter über verschiedene Ereignisse informiert.

Die GreenConfig-Parameter haben neben der für den Benutzer sichtbaren Modell-API die Funktion, im Service-Plug-in als zentrale Konfigurationseinheit gespeichert zu werden (vgl. folgenden Abschnitt 5.1.2). Jeder Modell-Parameter im System, d.h. auch die mit Adaptern auf andere Konfigurationsmechanismen abgebildeten, werden intern vom Adapter auf GreenConfig-Parameter umgesetzt, sodass sie im Service-Plug-in gleich behandelt werden können.

In diesem Kapitel werden weitere konzeptionell wichtige Merkmale der GreenConfig-Parameter vorgestellt.

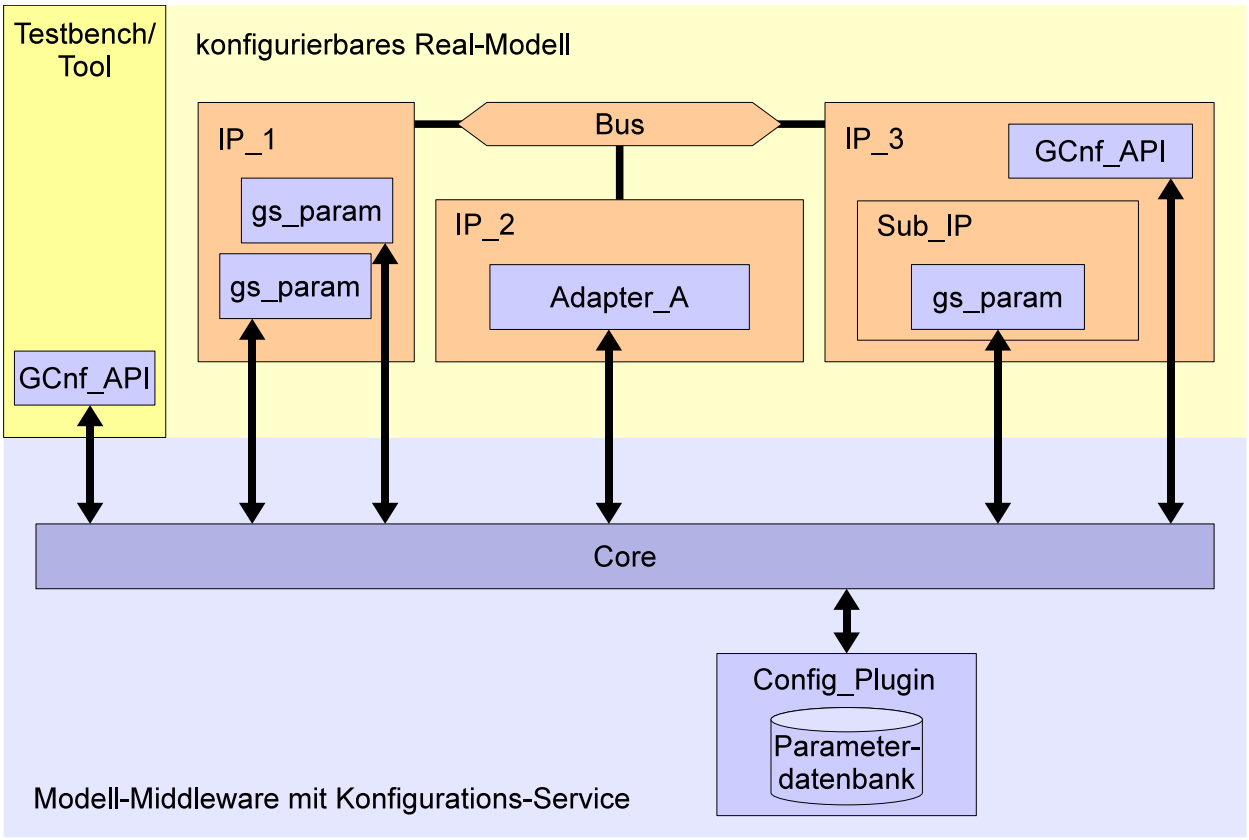


Abbildung 5.1.: Grundlegende Struktur des Konfigurations-Services in einer Simulation (Beispiel). Die gs_param-Blöcke und der Adapter_A sind Modell-APIs, die GCnf_APIs sind Tool-APIs.

5.1.2. Übersicht Konfigurations-Plug-in

Die zentrale Funktionalität des universellen Konfigurationsmechanismus GreenConfig ist im **Konfigurations-Plug-in** `ConfigPlugin` gekapselt. Dieses Service-Plug-in ist für den Endnutzer, der das Modell entwickelt oder ausführt, nicht sichtbar, da die Benutzerinteraktion mit den User-APIs stattfindet, die wiederum über die Modell-Middleware mit dem Konfigurations-Plug-in kommunizieren. Durch die erweiterbaren Transaktionen bleibt das Plug-in separat von existierenden User-APIs erweiterbar.

Die wichtigste Aufgabe des Konfigurations-Plug-ins ist die Verwaltung der Modell-Parameter-Datenbank. Es empfängt Transaktionen und setzt die Befehle in die entsprechenden Aufrufe in der Datenbank um. Das Interface `param_db_if` für die Anbindung der Datenbank erlaubt deren Austausch, sodass der Konfigurationsmechanismus für die Einbindung in externe Tools vorbereitet ist (vgl. Unterabschnitt „Austauschbare Datenbank“ in Abschnitt 5.5 auf Seite 110).

5.1.3. Übersicht GreenConfig-API

Die native Tool-API des Konfigurationsmechanismus ist die User-API **GreenConfig-API** (Klasse `GCnf_API`). Sie ist vorgesehen für den Zugriff auf Modell-Parameter von außerhalb

des besitzenden Moduls. Das sind konfigurierende Elemente wie die Testbench, die Entwicklungsumgebung, ein externes Tool (vgl. linker Kasten in Abbildung 5.1) oder auch ein anderes Modul, das beispielsweise ein Child-Modul konfiguriert (vgl. Modul IP_3 in Abbildung 5.1).

GreenConfig-API-Instanzen sollen vom Benutzer unter der Angabe des Modulpointers mit einer statischen Funktion abgerufen werden, d.h. es sollen keine Objekte selbst erzeugt werden. Somit kann der Konfigurationsmechanismus kontrollieren, welches GreenConfig-API-Objekt verwendet werden soll. Das ist beispielsweise für das Verstecken von Modell-Parametern und den Zugriff auf diese von Bedeutung.

Die GreenConfig-API ermöglicht dem Benutzer Zugriffe auf die Parameterdatenbank, beispielsweise das komfortable Abrufen von einzelnen Parametern oder ganzen Parameterlisten oder die Prüfung, ob Parameter existieren oder bereits verwendet wurden. Sie ermöglicht das Setzen von Initialwerten bevor der zugehörige Parameter existiert. Zudem gibt es die Möglichkeit, sich als Beobachter für neue Parameter zu registrieren.

5.1.4. Übersicht Adapter-APIs

Adapter-APIs sind weitere User-APIs des Konfigurations-Service. Sie nutzen die Stärken der Modell-Middleware und tragen zur Realisierung des Hauptziels der Austauschbarkeit von Modellen bei, indem sie die Integration von proprietären Konfigurationsmechanismen durchführen. Reale Adapter zu kommerziellen Entwicklungsumgebungen sind beispielsweise in den Abschnitten 6.1, 6.2, 6.3 und 6.5 beschrieben. Sie sind entweder für die Integration von fremden Konfigurationsmechanismen oder von fremden Modellen vorgesehen:

Adapter-APIs bieten eine API an, die der einer anderen möglichst genau entspricht, sodass sie stattdessen genutzt werden kann. Adapter-Modell-APIs sind dazu vorgesehen, von einem Modell, das für einen fremden Konfigurationsmechanismus geschrieben wurde, transparent wie der vom Modell erwartete Mechanismus verwendet zu werden (vgl. Adapter_A in Abbildung 5.1). Adapter-Tool-APIs bieten die Tool-API eines fremden Konfigurationsmechanismus an und übersetzen sie. Je nach Ähnlichkeit der fremden API kann der Adapter intern entweder eine existierende User-API verwenden, beispielsweise die GreenConfig-API, oder direkt mit den entsprechenden, möglicherweise erweiterten Middleware-Transaktionen und -Ports arbeiten. Eine Übersicht über mögliche Ansätze ist in Abschnitt 5.9 auf Seite 121 gegeben. Beispiele für Adapter-APIs werden in Kapitel 6 präsentiert.

5.2. Merkmale und Anforderungen

Dieser Abschnitt listet in Tabelle 5.2 diejenigen Merkmale von GreenConfig auf, die für die Erfüllung der Anforderungen aus Abschnitt 3.2 von Bedeutung sind. Sie werden kurz erläutert und teilweise mit Verweisen auf Details und die erfüllten Anforderung versehen. Da alle in Tabelle 3.3 auf Seite 36 definierten Merkmale unterstützt werden, wird auch Anforderung AA1 erfüllt. Die Merkmale sind mit industriellem Einfluss, z.B. von Intel und der CCI WG

entstanden. Im Vergleich zum Stand der Technik (Abschnitt 2.6) sind besonders die in den folgenden Unterabschnitten 5.2.1 und 5.2.2 beschriebenen Merkmale hevorzuheben.

Im Verlauf dieses Kapitels wird verdeutlicht, dass neben den in der Tabelle aufgelisteten Anforderungen alle verbleibenden Anforderungen nach Abschnitt 3.2 erfüllt sind.

5.2.1. Laufzeitkonfiguration

Ein besonderes Merkmal von GreenConfig ist die Laufzeitkonfiguration. Das bedeutet, dass Modell-Parameter während der gesamten Programmphase einschließlich der Simulationsphase auch vom Tool, also nicht nur dem Besitzer, konfiguriert werden können.

Dieses Merkmal ist ein Alleinstellungsmerkmal für Konfigurationsmechanismen der Related Work, die den Class-Wrapper-Ansatz verwenden (vgl. Tabelle 3.3 auf Seite 36). Da der Class-Wrapper-Ansatz mächtiger als der Konfigurations-Interface-Ansatz ist und der Konfigurations-Interface-Ansatz in den Class-Wrapper-Ansatz integriert werden kann (vgl. Abschnitt 3.3.3), hat GreenConfig somit das Potential, die meisten Merkmale zu unterstützen und bezüglich dieser Merkmale Adapter zu allen anderen Mechanismen bereitstellen zu können.

5.2.2. Integrationsfähigkeit

Ein weiteres besonders hervorzuhebendes Merkmal von GreenConfig ist die Integrationsfähigkeit. Allgemein sind darunter alle Elemente zu verstehen, die der Erfüllung des Hauptziels dieser Arbeit dienen, die Austauschbarkeit von Modellen zu erhöhen. Konkret sind das die in Tabelle 5.2 aufgelisteten Merkmale, die die Anforderungen AA2, 3, 4, 5, 6, 7 und KA4 erfüllen. Die Bereitsstellung von Adapter-User-APIs zu anderen Konfigurationsmechanismen ist ein Alleinstellungsmerkmal unter den Konfigurationsmechanismen (vgl. Abschnitte 2.6 und 3.1). Abschnitt 5.9 behandelt detailliert die Integrationsfähigkeit von GreenConfig.

5.3. Callbacks

Callbacks sind eine spezielle Art der Benachrichtigung (vgl. Seite 37). Es handelt sich um Aufrufe zuvor registrierter Beobachterfunktionen, die in den Funktionsparametern weiterführende Informationen übergeben bekommen können. Callbacks werden, im Gegensatz zu potentiell verzögerten SystemC-Events, unmittelbar und direkt ausgeführt. Das bedeutet für den Beobachter, dass er sofort auf die Benachrichtigung reagieren kann (je nach Callback auch mit einem entsprechenden Rückgabewert), das bedeutet aber auch, dass keine SystemC-Aktionen, die einen Kontextwechsel erforderlich machen, durchgeführt werden können[2].

[2]Das bedeutet konkret vor allem, dass keine SystemC-`wait`-Anweisungen erlaubt sind.

GreenConfig-Merkmal	**Details, Abschnitt**	**Anforderung**
unterstützt beide Konfigurationsansätze (F1)	Class-Wrapper und Konfigurations-Interface, siehe Abschnitt 5.9.1	AA2, AA7
unterstützt beide Konfigurationsrangfolgen (F14)	zeitlich und hierarchisch, siehe Abschnitt 5.9.2	AA3, AA7
Erweiterbarkeit durch flexible interne API	User-API-Konzept, Modell-Middleware GreenControl (siehe Kapitel 4)	AA4, AA5, AA7
C++-, SystemC-Datentypen (F2)	unterstützte Datentypen, siehe Tabelle 5.7	KA3
benutzerdefinierte Datentypen (F3)	vom Benutzer spezialisierbare Parameterklasse und beliebig ableitbare Basisklasse (Abschnitt 5.4)	KA3
Laufzeitkonfiguration (F9, F10)	siehe Abschnitt 5.2.1	KA1
Dateiausgabe (F4)	siehe Analyse-Service in Abschnitt 5.11	KA5
Dateieingabe	Konfigurationsdatei-Tool, siehe Abschnitt 5.8	KA6
Parameternamen	hierarchische und Top-Level-Namen, siehe Ab. 5.4.3	KA2
Parameterliste (F5)	Abruf von Parameterlisten (GreenConfig-API)	KA7
Parameterliste mit Wildcards (F13)	Abruf von Parameterlisten mit Modulnamen (GreenConfig-API)	KA14
Defaultwert (F6)	Konstruktor-Parameter des GreenConfig-Parameters (siehe Listing 5.5)	KA9
Initialwert (F7, F8, F10)	Setzen impliziter Parameter (vgl. Abschnitt 5.4.1) über GreenConfig-API	KA10
	Lesen impliziter Parameter über GreenConfig-API	KA11
typunabhängige Wertzugriffe	Zugriff auf (implizite und explizite) Parameter mit Strings, siehe Abschnitt 5.4.4	KA4
Parameterobjektwerte (F8, F9, F10)	Setzen und Lesen von Parameterobjektwerten	KA11, KA12
namensbasierter Zugriff auf Parameter (F8)	Zugriff auf (implizite und explizite) Parameter über GreenConfig-API	KA8
Benachrichtigungen (F11, 12)	verschiedene Callbacks, siehe Abschnitt 5.3	KA13
Parameter sperren	Sperren von Parametern, reject_write-Callbacks (Abschnitt 5.3) oder Sperren mit `lock` (Abschnitt 5.4.2)	KA16
Initialwert sperren	Sperren von Initialwerten (GreenConfig-API `lockInitValue`)	KA17
private GreenConfig-API	verstecken von Parametern, siehe Abschnitt 5.7	KA18
Verwendet-Abfrage	`is_used` in Parameterdatenbank und GreenConfig-API kontrolliert auf wertneutrale Zugriffe	KA19

Zusatzinformationen	in Basisparametern gespeicherte Dokumentation, `[set|get]_documentation`	KA20
Wert-Herkunft	optional durch Übergabe von Metadaten (Herkunft) in Funktionsaufrufen [3]	KA21

Tabelle 5.2.: Merkmale des Konfigurationsmechanismus GreenConfig; Merkmal-Nummern entsprechen Tabelle 3.3; Anforderungsnummern entsprechen Abschnitt 3.2

Der universelle Konfigurationsmechanismus verwendet Callbacks zum Benachrichtigen, da die unmittelbar erfolgenden Benachrichtigungen in jede andere Benachrichtigungsart, beispielsweise SystemC-Events, umgesetzt werden können, nicht jedoch umgekehrt.

GreenConfig unterstützt verschiedene Callback-Arten. Callbacks, die Parameterobjekte betreffen, werden direkt bei den Objekten registriert; objektunabhängige Callbacks werden bei der GreenConfig-API registriert. Die zur Verfügung stehenden Callback-Arten werden im Folgenden mit Beispiel-Use-Cases vorgestellt und sind als Aufzählung in Abbildung 5.4(b) auf Seite 102 zu sehen.

- *pre_read* – Callback vor dem Lesen eines GreenConfig-Parameterwerts: Dieser Callback wird durchgeführt, bevor ein Parameterwert gelesen wird. Somit kann der Wert vom Beobachter (z.B. einer Adapter-User-API) geändert werden, bevor er gelesen wird. Das ist für die Datensynchronisation zwischen verschiedenen Konfigurationsmechanismen notwendig (vgl. CASI in Abschnitt 6.3). Der GreenConfig-Parameterwert kann so nach Bedarf vor jedem Lesezugriff auf den Sollwert, der von einem anderen Mechanismus verwaltet wird, aktualisiert werden.

- *post_read* – Callback während des Lesens eines GreenConfig-Parameterwerts: Dieser Callback wird durchgeführt, während ein Parameterwert gelesen wird. Der Wert darf innerhalb dieses Callbacks nicht mehr geändert werden. Unten wird dieser Callback detaillierter analysiert. Dieser Callback kann zum Loggen von Lesezugriffen verwendet werden.

- *reject_write* – Callback vor dem Ändern eines GreenConfig-Parameterwerts mit der Möglichkeit zum Ablehnen: Der Callback wird durchgeführt, bevor ein Parameterwert geschrieben werden soll. Für den Beobachter besteht die Möglichkeit, über den Rückgabewert das Schreiben abzulehnen. Folglich ist bei diesem Aufruf nicht sicher, dass der Wert geändert werden wird (vgl. pre_write-Callback).

- *pre_write* – Callback vor dem Ändern eines GreenConfig-Parameterwerts: Der Callback wird durchgeführt, bevor ein Parameterwert geschrieben wird. Zu diesem Zeitpunkt ist bereits sicher, dass der Wert geschrieben werden wird, d.h. er kann nicht mehr zurückgewiesen werden. Ein Use-Case ist das Loggen von Wertänderungen, hier also zunächst das Aufzeichnen des alten Werts, beispielsweise zum Debuggen oder der Analyse.

[3] Die Übergabe von Metadaten ist hier nicht näher dokumentiert, siehe dazu [ScKl11a][i].

- *post_write* – Callback nach dem Ändern eines GreenConfig-Parameterwerts: Der Callback wird durchgeführt, nachdem der Parameterwert im Parameterobjekt geschrieben wurde. Dies kann wieder dem Debuggen oder der Analyse dienen, aber auch in Adaptern genutzt werden, um Parameteränderungen an einen anderen Mechanismus weiterzugeben.

- *destroy_param* – Callback beim Zerstören eines GreenConfig-Parameters: Der Callback teilt einem Beobachter mit, dass das Parameterobjekt im Anschluss nicht mehr zur Verfügung steht, da es zerstört wird. Das ermöglicht es einem Adapter, der beispielsweise Parameterobjekte in eine andere Darstellungsform überführt, diese auch dort zu entfernen.

- *create_param* – Callback beim Erzeugen eines neuen Parameters: Die GreenConfig-API benachrichtigt Beobachter mit diesem Callback über neu erstellte Modell-Parameter. Der Callback informiert sowohl über neue implizite Parameter als auch neue explizite Parameter (vgl. Abschnitt 5.4.1).

Eine Sonderstellung nimmt der in der obigen Liste auftretende Callback post_read ein: Um der Systematik gerecht zu werden, müsste es sich dabei um eine Benachrichtigung handeln, nachdem ein Parameterwert gelesen wurde. Bei diesem Callback-Typ treten zwei Probleme auf: Zum einen ist die Semantik schwierig festzulegen, zum anderen ist die technische, generische Realisierung eines tatsächlichen Callbacks nach dem Lesen schwierig oder unmöglich.

Es ist schwierig die *Semantik* zu definieren, wann der Zeitpunkt für die Benachrichtigung überhaupt sein soll:

Semantische Möglichkeit 1 Eine technisch naheliegende Möglichkeit für die Definition des Zeitpunkts ist innerhalb der lesenden Parameterfunktion. Semantisch problematisch ist hierbei, dass der Callback ausgelöst werden würde, bevor der Wert beim Aufrufer angekommen ist. Technisch problematisch ist, dass nicht sichergestellt werden kann, dass der Callback tatsächlich *nach* dem Lesevorgang auf dem Wert stattfindet[4].

Semantische Möglichkeit 2 Eine alternative Möglichkeit ist die Definition eines späteren Zeitpunkts: Soll der Callback später stattfinden, müsste er ausgeführt werden, nachdem der Aufrufer den Wert übernommen hat, möglicherweise auch erst, nachdem der Aufrufer ihn verarbeitet hat. Es bleibt unklar, wann das allgemeingültig sein soll. Zudem ist auch diese Alternative technisch problematisch: Nach der Rückkehr aus der lesenden Funktion zum Aufrufer ist es in C++ zunächst nicht mehr möglich, ohne Benutzerunterstützung einen Callback

[4]Das Problem ist hier, dass der tatsächliche Lesevorgang aus dem Wert in vielen Funktionen nur direkt innerhalb des `return`-Statements möglich ist. Es wäre allerdings wünschenswert, den Callback innerhalb der Lesefunktion nach diesem Lesevorgang auszulösen, wenn der Wert selbst ebenfalls Seiteneffekte bei einem Lesevorgang auslösen würde. Die Realisierung wäre nur innerhalb von Funktionen möglich, die Werte oder konstante Referenzen zurückgeben, die in einer temporären Variable gespeichert werden können – allerdings mit einem Performance-Verlust durch die notwendige Kopieroperation.

auszulösen. Eine generische Lösung wäre eine Verzögerung des Callbacks unter Ausnutzung von SystemC-Eigenschaften[5]. Zum einen würde das aber die Aufgabe der Unmittelbarkeit des Callbacks bedeuten, zum anderen würde das Konzept durchbrochen werden, nicht auf SystemC zu basieren. Zudem würde es zu seltsamem und schwierig zu behandelndem Verhalten führen, wenn mehrere, möglicherweise voneinander abhängige Zugriffe oder Änderungen auf einen oder mehrere Parameter gesammelt zu einem späteren Zeitpunkt stattfinden würden[6].

Realisierung Als Folge aus den semantischen und technischen Problemen ist der post_read-Callback ein Kompromiss: Er wird innerhalb der Lesefunktion im Parameter ausgelöst, also bevor der Wert vom Aufrufer übernommen wurde. Es wird nicht festgelegt, ob der Wert zum Zeitpunkt des Callbacks tatsächlich gelesen wurde. Der Callback unterscheidet sich folglich vom pre_read-Callback nur darin, dass der Wert vom Beobachter nicht mehr geändert werden darf.

5.4. GreenConfig-Parameter

GreenConfig-Parameter sind die Modell-Parameter des GreenConfig-Konfigurationsframeworks und damit eine Modell-API. Es handelt sind um Class-Wrapper, die den zugehörigen Datentyp transparent ersetzen. Der Parameterbesitzer, also beispielsweise ein SystemC-Modul, instanziiert ein Parameterobjekt (Typ `gs_param<Datentyp>`) mit dem gewünschten Datentyp. Dieses Objekt soll im Besitzer so lange für den lokalen Zugriff existieren, wie der Modell-Parameter benötigt wird. Das Objekt registriert sich automatisch bei der Datenbank im Konfigurations-Plug-in.

Die Parameter-Basisklasse (`gs_param_base`) erlaubt den eingeschränkten Zugriff ohne Kenntnis des Datentyps. Die Basisklasse kann außerhalb des Besitzers vom Konfigurationsmechanismus, dem Tool oder anderen Modulen zum Speichern und Weitergeben des Parameters verwendet werden. Auch das Umwandeln (Cast) in den tatsächlichen GreenConfig-Parametertyp kann vorgenommen werden.

Die Anforderung an Modell-Parameter speziellen Typs ist, dass sie von dieser Basisklasse ableiten müssen. Die meisten in GreenConfig existierenden, davon abgeleiteten Klassen dienen der Bequemlichkeit und werden von GreenConfig speziell unterstützt (beispielsweise die typisierte Parameterklasse `gs_param`). Sie bieten einheitliche Schnittstellen und einfache Anpassungsfähigkeit an neue Datentypen, müssen aber nicht für neue Spezialisierungen verwendet werden.

[5] Beispielsweise könnte der Callback nach einem Delta-Schritt ausgelöst werden. Damit wäre sichergestellt, dass der aufrufende Prozess den Wert übernommen hat und die Kontrolle abgegeben hat.

[6] Es könnten problematische Race-Conditions auftreten

5.4.1. Implizite und explizite Parameter

Wie bereits bekannt, verwendet der Konfigurationsmechanismus GreenConfig den Class-Wrapper-Ansatz. GreenConfig verwendet für einen Modell-Parameter (vgl. Definition auf Seite 37) genau ein zugehöriges Parameterobjekt. Ein Modell-Parameter, der (z.B. über die GreenConfig-API) einen Initialwert zugewiesen bekommt, bevor das Parameterobjekt vom Besitzer erzeugt wird, heißt **impliziter Parameter**. Da der Datentyp des Parameters noch nicht notwendigerweise bekannt ist, können Werte impliziter Parameter nur als Zeichenkette gesetzt werden.

Sobald ein Modell-Parameter als `gs_param`-Objekt erzeugt wird, ist er ein **expliziter Parameter**. Existiert für den entsprechenden Parameternamen bereits ein impliziter Parameter, wird dessen Wert übernommen, und der Parameter wird explizit. Beim Konstruieren eines GreenConfig-Parameters kann im Konstruktor optional ein Defaultwert angegeben werden. Dieser wird von einem potentiell vorhandenen Initialwert sofort überschrieben. Gibt es weder einen Initial- noch einen Defaultwert, nimmt der Parameter einen Standardwert an[7].

Die Umwandlung von impliziten Parameterwerten in einen expliziten Parameter wird mit einer Deserialisierung der Zeichenkette vorgenommen, die vom `gs_param`-Parameterobjekt bereitgestellt wird. Auch der umgekehrte Weg, die Serialisierung eines expliziten Parameters in eine Zeichenkette, die den Parameterwert repräsentiert, wird von der Parameterklasse bereitgestellt.

Als C++-Objekt kann ein expliziter Parameter auch zerstört werden. Falls es vor dessen Existenz bereits einen impliziten Parameter gab, wird dieser wieder mit dem ursprünglichen Initialwert aktiv.

Die Präzedenzen der verschiedenen Werte verhalten sich wie in Abschnitt 3.1.3 vorgestellt.

5.4.2. Klassenhierarchie

Abbildung 5.3 zeigt die Klassenhierarchie der GreenConfig-Parameter. Weitere direkt mit den Parametern in Zusammenhang stehende Klassen werden in Abbildung 5.4 zusammengefasst.

Basisklasse

Die abstrakte Basisklasse `gs_param_base` hängt nicht vom Datentyp des Parameters ab und stellt sämtliche typunabhängige Funktionalität zur Verfügung. Diese Klasse dient der Speicherung und Weitergabe von GreenConfig-Parametern. Sämtliche Operationen des Konfigurations-Services finden nur auf dieser Klasse, nicht auf den abgeleiteten Spezialisierungen statt. Folglich ist es für spezielle Modell-Parameter nicht zwingend Voraussetzung, dass sie

[7] Typischerweise ist der Standardwert der des leeren Konstruktors des Datentyps, den der Parameter repräsentiert. Die Template-Spezialisierung der `gs_param`-Klasse kann allerdings den Standardwert z.B. bei komplexen Datentypen beliebig setzen.

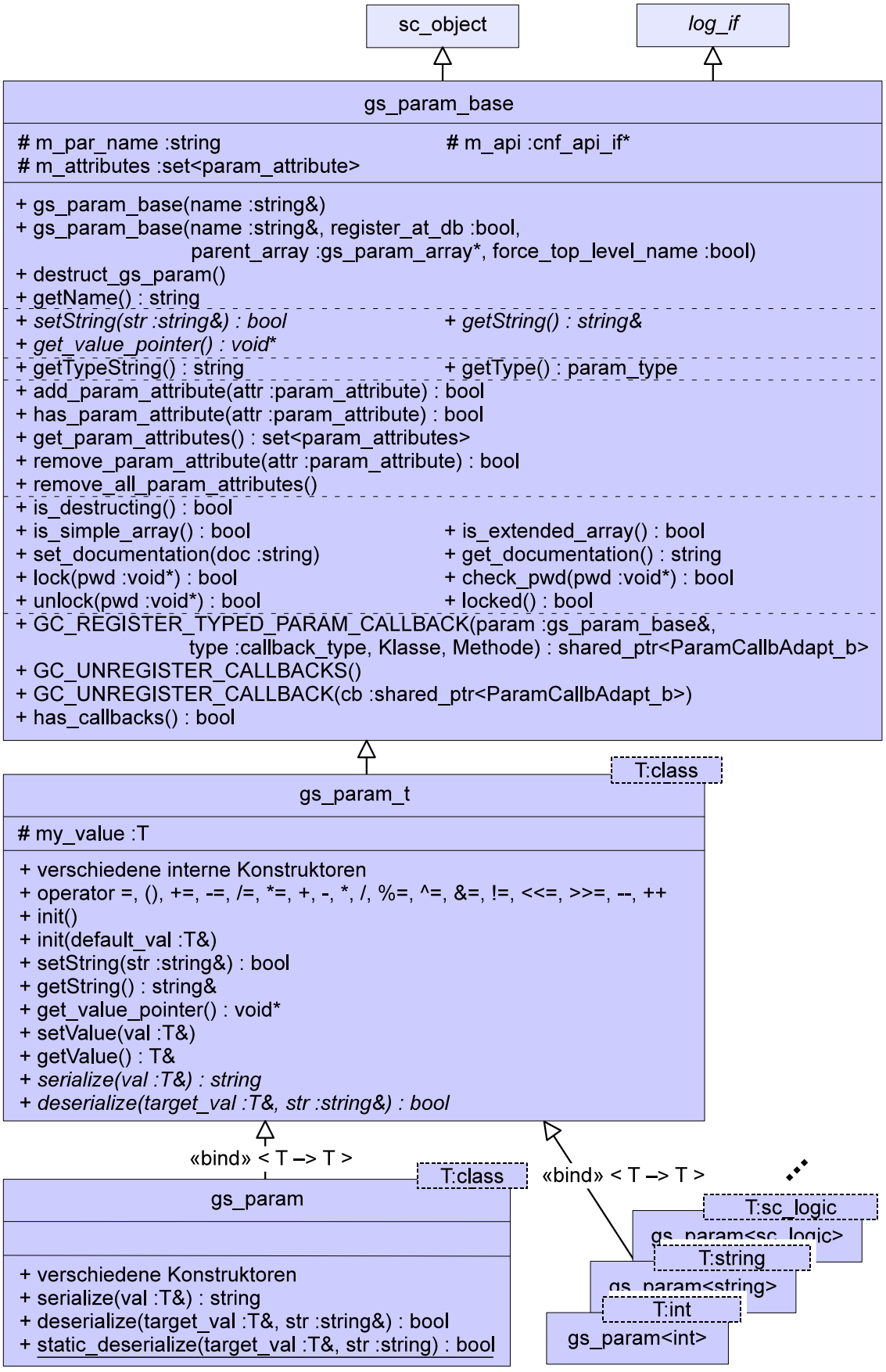
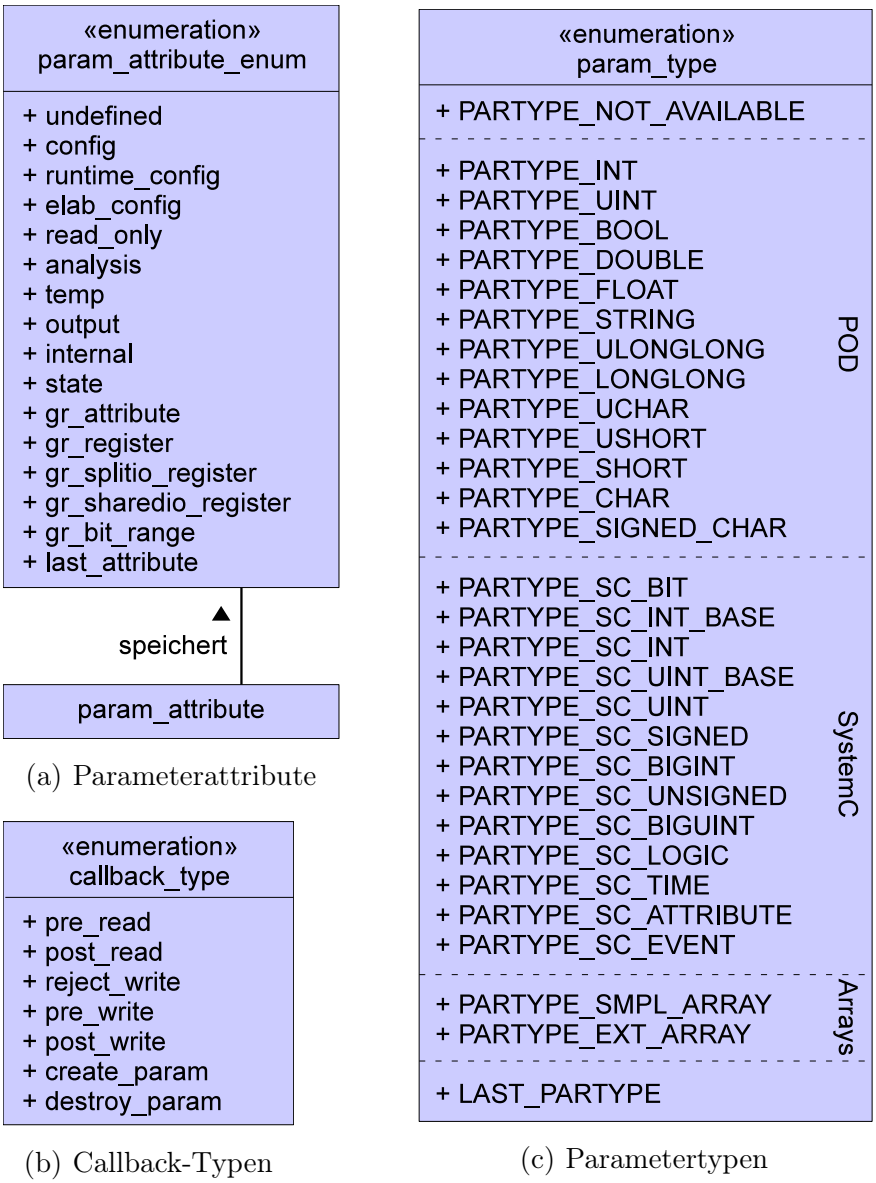


Abbildung 5.3.: GreenConfig-Parameter (Klassendiagramm)



(a) Parameterattribute

(b) Callback-Typen

(c) Parametertypen

Abbildung 5.4.: Unterstützende Datentypen für GreenConfig-Parameter (Klassendiagramm)

auf der von der Basisklasse abgeleiteten, in Abbildung 5.3 gezeigten Struktur aufbauen, sondern lediglich, dass sie von der Basisklasse ableiten.

Der *Konstruktor* der Basisklasse kann mit nur einem Parameternamen oder mit weiteren Angaben aufgerufen werden: Der Parametername ist per Default ein lokaler Name, dem der Modulname des besitzenden Moduls vorangestellt wird. Der Name wird im Member `m_par_name` gespeichert und kann mit der Funktion `getName` ausgelesen werden. Das Setzen des Konstruktor-Parameters `force_top_level_name` ermöglicht das direkte Setzen eines beliebigen (auch hierarchischen, d.h. Punkte enthaltenen) Namens. Ist der Parameter Teil eines Parameter-Arrays, muss das übergeordnete Array im Konstruktor-Parameter `parent_array` übergeben werden. Der Parameter `register_at_db` erlaubt das Unterdrücken der automa-

tischen Registrierung des GreenConfig-Parameters beim Service-Plug-in[8]. Der Konstruktor holt sich die für diesen Parameter zuständige GreenConfig-API und speichert den Pointer im Member `m_api`. Der Parameterwert wird nicht in der Basisklasse gespeichert, sondern in einer abgeleiteten als Wert richtigen Typs.

Einige *rein virtuelle Funktionen* ermöglichen den Zugriff auf den Wert: Die String-Funktionen `setString` und `getString` fordern von der abgeleiteten Klasse die Konvertierung des Werts in eine Zeichenkette (vgl. Abschnitt 5.4.4 auf Seite 107). Dagegen soll die Funktion `get_value_pointer` den Wert als `void`-Pointer liefern.

Der Typ eines Parameters heißt **GreenConfig-Parameter-Typ** und zeigt den Datentyp an, den der Parameter transparent repräsentiert. Er wird entweder als Aufzählung vom Typ `param_type` (vgl. Abbildung 5.4(c)) oder als Zeichenkette angegeben. Die Aufzählung deckt die im Framework bereits vorhandenen Datentypen ab, die Zeichenkette kann von weiteren Spezialisierungen beliebig (aber eindeutig) vergeben werden. Die virtuellen Typ-Funktionen `getType` und `getTypeString` liefern den Typ des Parameters als Zeichenkette oder als Aufzählung, sofern diese Informationen verfügbar sind. Diese Funktionen sollen von den Spezialisierungen überschrieben werden.

Einem Parameter können beliebig viele **GreenConfig-Parameter-Attribute** zugewiesen werden. Es handelt sich hierbei um zunächst rein informative Einordnungen des Modell-Parameters in Kategorien. Ein Konfigurator- oder Analyse-Tool könnte Parameter entsprechend dieser Markierung unterschiedlich behandeln oder sortieren. Attribute sind vom Typ `param_attribute`, der die vordefinierten Attribute der Aufzählung `param_attribute_enum` oder benutzerdefinierte speichert (vgl. Abbildung 5.4(a)). Die Organisation dieser Attribute findet im Parameter über die `*param_attribute`-Funktionen statt und wird im Member `m_attributes` gespeichert.

Die folgenden Funktionen liefern Informationen über den *Parameterzustand*: Von der Funktion `is_destructing` wird „wahr“ geliefert, wenn das Parameterobjekt gerade zerstört wird. Das ist der Fall, wenn der Destruktor der niedrigsten abgeleiteten Klasse die Funktion `destruct_gs_param` aufruft, wozu er verpflichtet ist. Die Statusabfrage ist für einen Beobachter interessant, der sich für Callbacks registriert hat. Zwei weitere Funktionen liefern Informationen, ob es sich bei diesem Parameter um ein einfaches oder erweitertes Array handelt (`is_[simple|extended]_array`, vgl. Abschnitt 5.4.5).

Die beiden Funktionen `[set|get]_documentation` setzen und liefern menschenlesbare Zusatzinformationen über den Parameter, beispielsweise Hinweise über erlaubte Werte, deren Einheit oder lose Abhängigkeiten zu anderen Parametern. Diese Informationen können beispielsweise vom besitzenden Modul angegeben werden.

Die Basisklasse unterstützt das Sperren expliziter Parameter: Die Funktion `lock` sperrt den Parmeter gegen alle manipulativen Zugriffe. Optional kann ein Passwort in Form ei-

[8] Dadurch ist es möglich, die Vorteile eines GreenConfig-Parameterobjekts außerhalb des Konfigurationsmechanismus zu nutzen – dies dient *nicht* dem Verstecken von Modell-Parametern (vgl. Abschnitt 5.7).

ner Pointeradresse angegeben werden, das auch zum Entsperren mit der Funktion `unlock` benötigt wird.

Die Callbacks für Ereignisse am Parameterobjekt (vgl. Abschnitt 5.3 auf Seite 95) können für Methoden mit der folgenden Signatur registriert werden:

```
callback_return_type config_callback(gs_param_base& chngd_par, callback_type ty);
```

Zum Registrieren wird das Makro `GC_REGISTER_TYPED_PARAM_CALLBACK` verwendet. Der zurückgegebene Pointer kann für das Löschen des Callbacks verwendet werden: Ausgewählte Callbacks können mit dem Makro `GC_UNREGISTER_CALLBACK` entfernt werden, das Makro `GC_UNREGISTER_CALLBACKS` entfernt alle Callbacks wieder. Die Funktion `has_callbacks()` gibt an, ob Callbacks bei diesem Parameterobjekt registriert sind.

Typisierte Zwischenklasse

Die Klasse `gs_param_t` ist eine typisierte Zwischenklasse, die von der Basisklasse ableitet. Sie implementiert diejenigen Funktionen, die in C++ zwar typabhängig sind, aber nicht explizit spezialisiert werden müssen, d.h. für jeden Datentyp gleich sein können.

Die verschiedenen Konstruktoren entsprechen im Wesentlichen denen der Basisklasse und geben die Parameter an deren Konstruktoren weiter. Der Member `my_value` speichert den tatsächlichen Parameterwert. Die beiden Initialisierungsfunktionen `init` registrieren den Parameter beim Konfigurations-Plug-in (wenn nicht per Konstruktor-Parameter unterbunden) und setzen den Parameter auf den Defaultwert. Einige Operatoren werden nur deklariert, sodass sie später von Spezialisierungen implementiert werden können.

In dieser Zwischenklasse werden die oben erwähnten String-Funktionen und die Pointer-Funktion `get_value_pointer` für den typunabhängigen Zugriff implementiert, sodass sie per virtuellem Aufruf aus der Basisklasse erreichbar sind. Die Funktionen `setValue` und `getValue` für den Wertzugriff, sowie die Gleichheits- und Klammeroperatoren manipulieren den Parameterwert jeweils unter Beachtung der auszuführenden Callbacks.

Die verbleibenden Konvertierungsfunktionen nehmen die Umwandlung des Parameterwerts von und in Zeichenketten vor und werden im Abschnitt 5.4.4 auf Seite 107 erläutert.

Typisierte Parameterklasse

Die GreenConfig-Parameterklasse `gs_param` ist die vom Endnutzer als Modell-Parameter zu verwendende Klasse. Wenn keine explizite Spezialisierung vorhanden ist, wird die generische Realisierung verwendet, ansonsten eine der vielfältig vorhandenen Template-Spezialisierungen für die entsprechenden Datentypen.

Die Parameterklasse leitet von der Zwischenklasse `gs_param_t` ab, wodurch schon einige typabhängige Funktionalität vorhanden ist. Lediglich die Konstruktoren und die für jeden Datentyp anders zu implementierenden Funktionen müssen hier bereitgestellt werden. Im allgemeinen Fall sind das die Konvertierungsfunktionen, die nur in den Fällen funktionieren,

in denen der Datentyp T den leeren Konstruktor sowie die Streaming-Operatoren zur Verfügung stellt. Die Kontruktoren der GreenConfig-Parameter (Klasse `gs_param<Typ>`) sind in Listing 5.5 aufgeführt. Für verschiedene und flexible Anwendungsgebiete gibt es verschieden umfangreiche Kontruktorparameterlisten: Der einfachste Fall ist die Angabe eines lokalen Namens, möglicherweise mit Angabe eines Defaultwerts als Zeichenkette oder Typ. Die Angabe, ob der übergebene Name ein lokaler ist, also mit dem Modulnamen des besitzenden Moduls automatisch erweitert wird, ist mit dem Parameter `force_top_level_name` möglich. Der Standardfall ist ein lokaler Name. Wird der Parameter als Teil eines Arrays erstellt, muss das übergeordnete Array im Parameter `parent_array` übergeben werden. Nur für Ausnahmefälle ist der Parameter `register_at_db` vorgesehen, der, wenn auf „falsch“ gesetzt, verhindert, dass sich der GreenConfig-Parameter beim Konfigurations-Plug-in registriert.

Spezialisierungen

Spezialisierungen der GreenConfig-Parameterklasse `gs_param` implementieren zusätzlich zu den Konvertierungsfunktionen die Funktionen, die den GreenConfig-Parameter-Typ liefern. Weichen die Eigenschaften des verwendeten Datentyps erheblich von den einfachen Standardtypen ab, kann eine vollständig neue Implementierung der Zwischen- und Parameterklasse sinnvoll sein. Der GreenConfig-Parameter `gs_param<sc_attribute<T> >` ist ein Beispiel für so einen Fall. Ein Beispiel für einen Parameter, der sogar einen anderen Klassennamen hat, ist das erweiterte Array `gs_param_array`. Die vom Konfigigurationsmechanismus direkt unterstützen Parameter-Spezialisierungen sind in Tabelle 5.7 auf Seite 109 gelistet.

5.4.3. Parameternamen

Um die Portabilität nicht einzuschränken, existieren generell keine Einschränkungen für Parameternamen. Ein Modell-Parameter wird über seinen vollständigen und systemweit eindeutigen Namen identifiziert. Zu diesem Zweck kennt das Konfigurations-Plug-in keine Hierarchie.

Für die eigenen vom Benutzer im Standardfall erstellten GreenConfig-Parameter werden jedoch die von SystemC-Modulen und -Objekten bekannten hierarchischen Namen verwendet (vgl. [IEEE06]). Das bedeutet, dass alle nativen GreenConfig-User-APIs den vollständigen Namen logisch in eine Hierarchie zerlegen und als Separator den Punkt verwenden. Die Parameterobjekte generieren ihren Namen automatisch aus dem SystemC-Namen ihres Besitzers und dem im Konstruktor übergebenen Namen[9]. Die GreenConfig-API erlaubt eine hierarchiebewusste Suche.

[9]Eine Ausnahme ist die Konstruktor-Option für einen vom Besitzer vorgegebenen Top-Level-Namen.

```
// Konstruktoren mit lokalem Namen
gs_param(const [string&|char*] nam)

// Konstruktoren mit (lokalem oder Top-Level-) Namen und Defaultwert als Zeichenkette
gs_param(const [string&|char*] nam, const [string&|char*] val, const bool force_top_level_name = false)

// Konstruktoren mit (lokalem oder Top-Level-) Namen und Defaultwert als Typ
gs_param(const [string&|char*] nam, const val_type &val, const bool force_top_level_name = false)

// Konstruktoren mit (lokalem oder Top-Level-) Namen, optionalem Defaultwert (als Zeichenkette oder Typ)
// und uebergeordnetem Array (als Referenz oder Pointer)
gs_param(                                                gs_param_array[*|&] parent_array, const bool force_top_level_name = false)
gs_param(const [string&|char*] nam                     , gs_param_array[*|&] parent_array, const bool force_top_level_name = false)
gs_param(const [string&|char*] nam, const [string&|char*] val, gs_param_array[*|&] parent_array, const bool force_top_level_name = false)
gs_param(const [string&|char*] nam, const val_type&        val, gs_param_array[*|&] parent_array, const bool force_top_level_name = false)

// Konstruktoren mit (lokalem oder Top-Level-) Namen, Defaultwert (als Zeichenkette), uebergeordnetem Array (als Referenz oder Pointer)
// und Option, ob der Parameter beim Service registriert werden soll
gs_param(const string& nam, const string& val, gs_param_array* parent_array, const bool force_top_level_name, const bool register_at_db)
gs_param(const string& nam, const string& val, gs_param_array& parent_array, const bool force_top_level_name, const bool register_at_db)
```

Listing 5.5: Konstruktoren der GreenConfig-Parameter (Klasse `gs_param<val_type>`))
(bei Angabe von [<option1> | <option2>] jeweils alle Kombinationen)
(val_type ist der Datentyp des Parameters)

Wie Abbildung 5.3 auf Seite 101 zeigt, sind GreenConfig-Parameter SystemC-Objekte, da sie von der Klasse `sc_object` ableiten. Dadurch können sie ihr besitzendes Parent-Modul identifizieren und dessen Namen für die Bildung des eigenen Parameternamens verwenden. Die Anforderung KA15, den Konfigurationsmechanismus in SystemC möglichst unauffällig zu halten, bleibt hiermit dennoch erfüllt[10].

Das Beispielmodell in Abbildung 5.6 zeigt den Unterschied zwischen automatisch benannten Parametern, die als Präfix den Namen des besitzenden Moduls bekommen, und Top-Level-Parameternamen, die entweder keine Hierarchie widerspiegeln (Parameter any_name und top_level_name) oder eine beliebige nicht existierende Hierarchie beinhalten können (zum Beispiel der Parameter MyNotExistingMod.SubMod.param). Das Objektdiagramm zeigt die Modulinstanzen mit den Parametern und als Anmerkungen die jeweils verwendeten Parameterkontruktoren. Die Konstruktoren demonstrieren die Verwendung und Auswirkung des `force_top_level_name`-Konstruktorparameters. Im unteren Teil ist die resultierende Parameterliste in der Datenbank des Konfigurations-Plug-ins abgebildet.

5.4.4. Typunabhängige Zeichenkettenzugriffe

Um Parameterobjekte soweit wie möglich typunabhängig behandeln zu können, werden die String-Funktionen der Basisklasse bereitgestellt (`setString`, `getString`). Sie sind rein virtuell und werden in der typisierten Klasse implementiert. Intern greifen sie auf die rein virtuellen, typabhängigen Funktionen `deserialize` und `serialize` der typisierten Zwischenklasse zu, die in der Parameterklasse implementiert werden. Um zunächst unabhängig von Template-Spezialisierungen zu sein, existiert eine generische Implementierung, die die Streaming-Operatoren verwendet. Das ist der Ursprung der Einschränkung für direkt, d.h. ohne explizite Template-Spezialisierung, unterstützte Datentypen.

Jede Template-Spezialisierung kann für den eigenen Datentyp die Syntax der Zeichenkette des Wertes selbst bestimmen.

5.4.5. Weitere Parametermerkmale

In diesem Abschnitt werden einige weitere wichtige interessante Merkmale kurz erläutert.

- *Parameterdatentypen*: Tabelle 5.7 auf Seite 109 listet die vom Konfigurationsmechanismus GreenConfig unterstützten Parameterdatentypen auf.

- *Arrays (Simple- und Extended-)*: Es werden zwei verschiedene Arten von Parameter-Arrays angeboten, deren Members GreenConfig-Parameter sind und – wie das Array selbst – in der Parameterdatenbank geführt werden. Tabelle 5.8 auf Seite 109 zeigt die Eigenschaften in einer Gegenüberstellung.

[10] SystemC-Objekte haben im Gegensatz zu SystemC-Modulen keine strukturelle Bedeutung und sind bei Analysen kaum zu beachten.

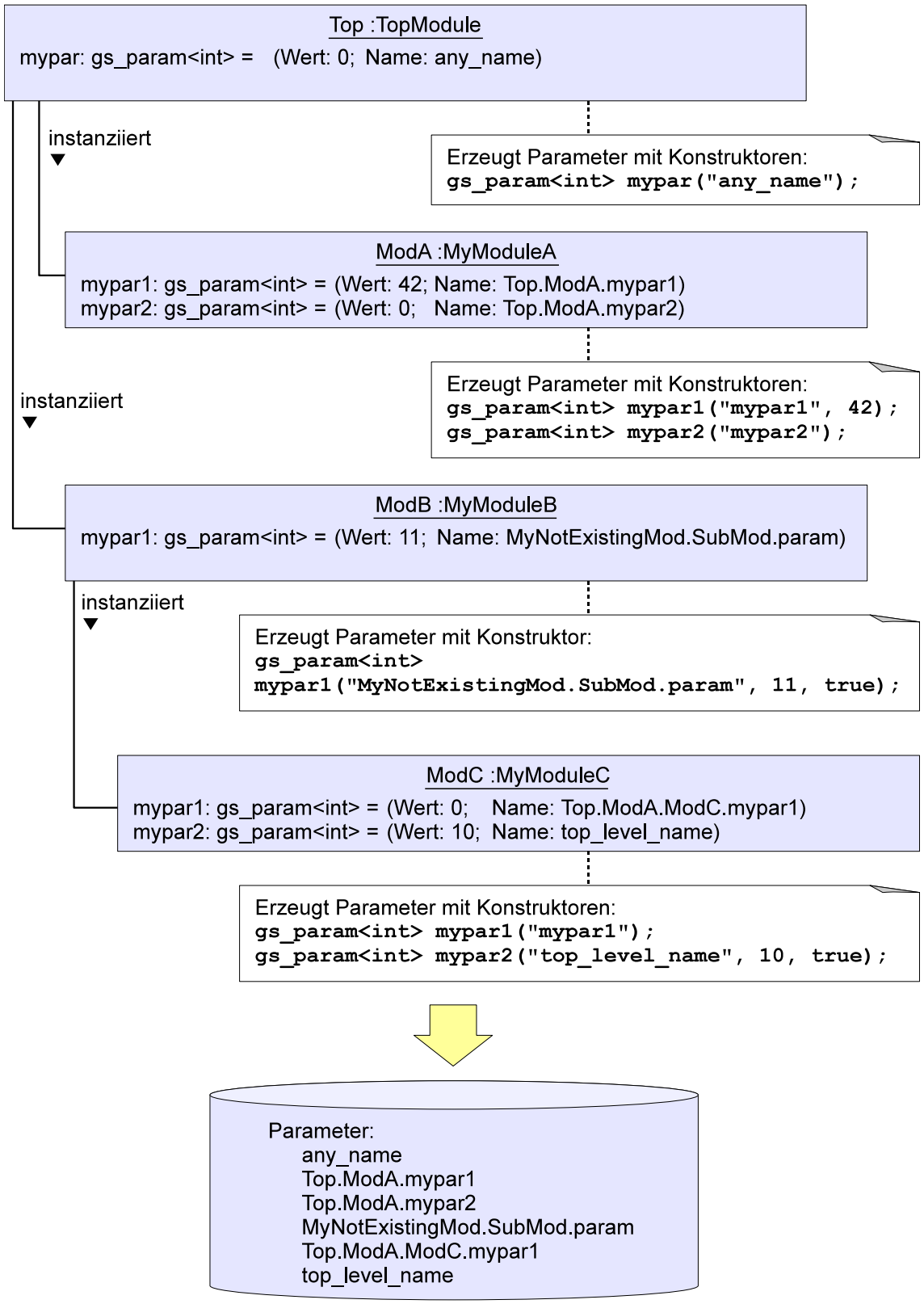


Abbildung 5.6.: Parameternamen-Beispiel: Beispiel-Module mit GreenConfig-Parametern und den entsprechend verwendeten Konstruktoren; resultierende Parameterdatenbank im Konfigurations-Plug-in

Kategorie	**Datentypen**
C++ POD	`int`, `unsigned int`, `bool`, `double`, `float`, `string`, `long long`, `unsigned char`, `signed char`, `char`, `unsigned short`, `short`
SystemC	`sc_int_base`, `sc_int<w>`, `sc_uint_base`, `sc_uint<w>`, `sc_signed`, `sc_bigint<w>`, `sc_unsigned`, `sc_biguint<w>`, `sc_bit`, `sc_logic`, `sc_time`, `sc_event`
Arrays	einfache: `template<class T> T*`, erweiterte: `gs_param_array`
generisch	Alle Datentypen, die den leeren Konstruktor und die beiden Streaming-Operatoren implementieren.
speziell	`std::vector<std::string>`

Tabelle 5.7.: Unterstützte GreenConfig-Parameterdatentypen

Eigenschaft	**Simple-Parameter-Array** (SPA)	**Extended-Parameter-Array** (EPA)
Klasse	`template<class T> gs_param<T*>`	`gs_param_array`
Members	alle vom gleichen Datentyp	können unterschiedlichen Datentyps sein
Parametername	nummeriert, nicht benannt	benannt
Zugriff	namesbasiert (z.B. arrayname.3) oder Vektor-ähnlich auf Parameterobjekt	namensbasiert (z.B. arrayname.membername) oder namensbasierte get-Funktion auf Objekt
Größe	Array-Größe kann vom Tool bestimmt werden	Array-Größe und Erzeugung der Members nur durch besitzendes Modul
Besonderheiten		verschachtelte Arrays möglich; Callback-Verhalten kann beeinflusst werden

Tabelle 5.8.: Parameter-Arrays (Simple- und Extended-)

- *Neue Parameterdatentypen anlegen*: Bisher nicht unterstützte Parameterdatentypen können durch den Endnutzer einfach durch Erstellen einer neuen Template-Spezialisierung der `gs_param`-Klasse realisiert werden. Dafür enthält [ScKl11a]^i eine Anleitung. Alternativ kann eine von den gegebenen typisierten Klassen unabhängige Implementierung gewählt werden, die lediglich von der Basisklasse `gs_param_base` ableitet.

5.5. Konfigurations-Plug-in

Das Konfigurations-Plug-in ist ein zentrales Singleton-Objekt und implementiert den Konfigurations-Service. Ein zentrales Plug-in ist sinnvoll für die Erfüllung der Anforderung nach einem globalen Verzeichnis (KA7).

Ansätze für das Speichern des Parameterwerts

Im Folgenden sollen zunächst Ansätze zur Speicherung der Parameterwerte diskutiert werden. Der hier entwickelte Konfigurationsmechanismus soll universell einsetzbar sein, also Adapter zu anderen Konfigurationsmechanismen ermöglichen. Damit muss die Wertrepräsentation in der Datenbank flexibel sein.

Ein Ansatz zum Speichern von Parameterwerten ist das direkte Speichern der Werte in einer Datenbank[11]. Die Darstellung dieser Werte muss von universellem Typ sein (vgl. Anforderung KA4). Universell ist die Speicherung als Zeichenkette, für die es eine Konvertierungsvorschrift in den tatsächlichen Parameterdatentyp geben muss, wenn dieser keine Zeichenkette ist. Alternativ und ebenfalls universell ist die Speicherung einer Wert-Basisklasse.

Wenn der Parameterwert auf diese Weise als Zeichenkette oder Wert-Basisklasse gespeichert werden soll, werden die daran gestellten Anforderungen bereits von den GreenConfig-Parametern vollständig erfüllt. Deswegen ist ein Ansatz denkbar, der die GreenConfig-Parameter für die Speicherung der Werte verwendet. Da die Modell-Parameter – im Fall von Modellen, die unter Verwendung des GreenConfig-Konfigurationsmechanismus geschrieben sind – bereits als GreenConfig-Parameter existieren, bietet es sich an, die Werte in der Datenbank als Verweise auf die Parameterobjekte zu speichern. Dieser Ansatz ist für GreenConfig gewählt worden.

Adapter zu anderen Konfigurationsmechanismen können demnach je nach Realisierung der Umsetzung die fremden Modell-Parameter auf GreenConfig-Parametern aufbauen (d.h. z.B. von ihnen ableiten oder sie als Member erzeugen) oder Instanzen von GreenConfig-Parametern innerhalb des Adapters verwalten und die Werte zum fremden Mechanismus spiegeln.

[11] Der Begriff Datenbank ist in diesem Kontext sehr allgemein zu verstehen und kann pragmatisch als assoziativer Container implementiert werden.

Der zu einem Modell-Parameter gehörende Datensatz kann im Plug-in um zusätzliche Informationen erweitert werden. Das sind beispielsweise Informationen über den impliziten Parameter: der Initialwert als Zeichenkette und Sperrinformationen des Initialwerts.

Aufbau Plug-in

Im Folgenden wird der Aufbau des Plug-ins beschrieben. Abbildung 5.9 zeigt im Zentrum die Klasse `ConfigPlugin` des Konfigurations-Plug-ins. Die statische Funktion `get_instance` kann verwendet werden, um den Service zu aktivieren, da diese Funktion das Plug-in-Singleton-Objekt erzeugt, wenn es noch nicht existiert. Für die Verbindung zum Middleware-Core leitet die Klasse vom `gc_port_if`-Interface ab, empfängt Transaktionen in der Funktion `transport` und hat den Member `m_gc_port`. Für das Kommandoattribut in den Transaktionen verwendet sie die Aufzählung `ConfigCommand`.

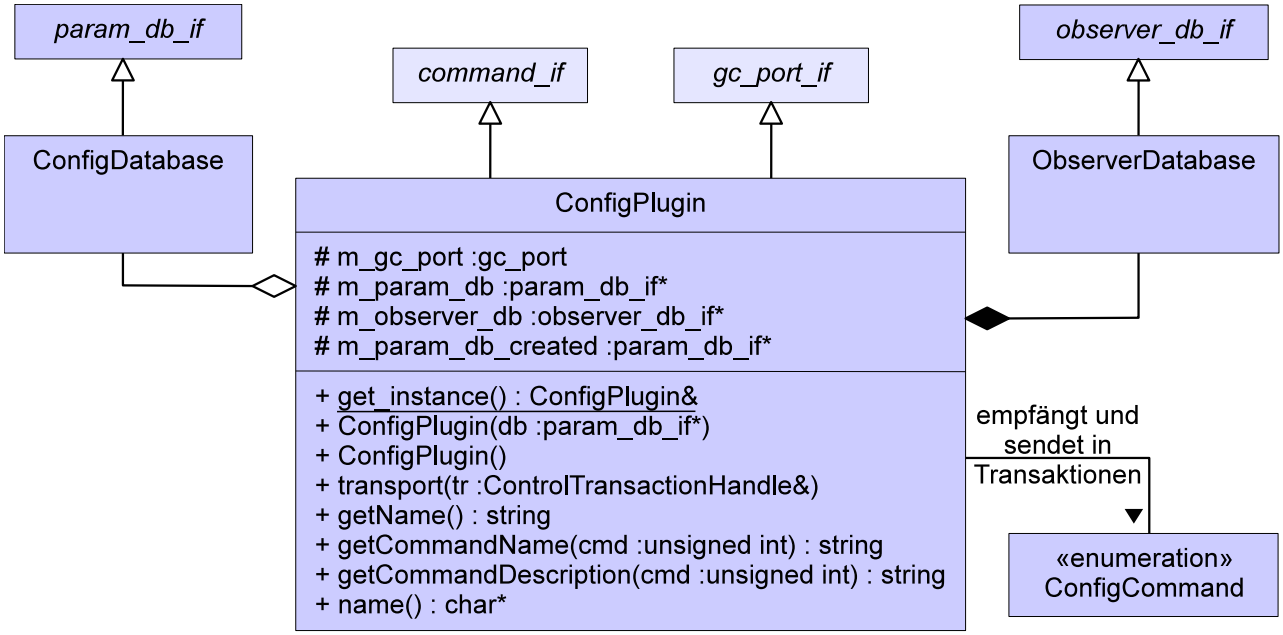


Abbildung 5.9.: Übersicht Konfigurations-Plug-in (Klassendiagramm)

Die Plug-in-Klasse besitzt zwei Konstruktoren, von denen einer die Übergabe einer vom Normalfall abweichenden Parameterdatenbank erlaubt. Das ermöglicht eine Variante der UIP-Integration aus Abschnitt 5.9.3. Der leere Konstruktor erzeugt die normale interne Parameterdatenbank vom Typ `ConfigDatabase`. Die Parameterdatenbanken müssen vom Interface `param_db_if` (siehe Abbildung 5.10) ableiten, damit sie unabhängig von der Realisierung an das Plug-in übergeben und dort verwendet werden können.

Zusätzlich zur variablen Parameterdatenbank besitzt die Plug-in-Klasse eine Beobachterdatenbank (Klasse `ObserverDatabase` mit dem Interface `observer_db_if`, siehe Abbildung 5.11). Sie speichert Beobachter-User-APIs (in Form ihrer Adresse), die an Callbacks über neu erzeugte Modell-Parameter interessiert sind.

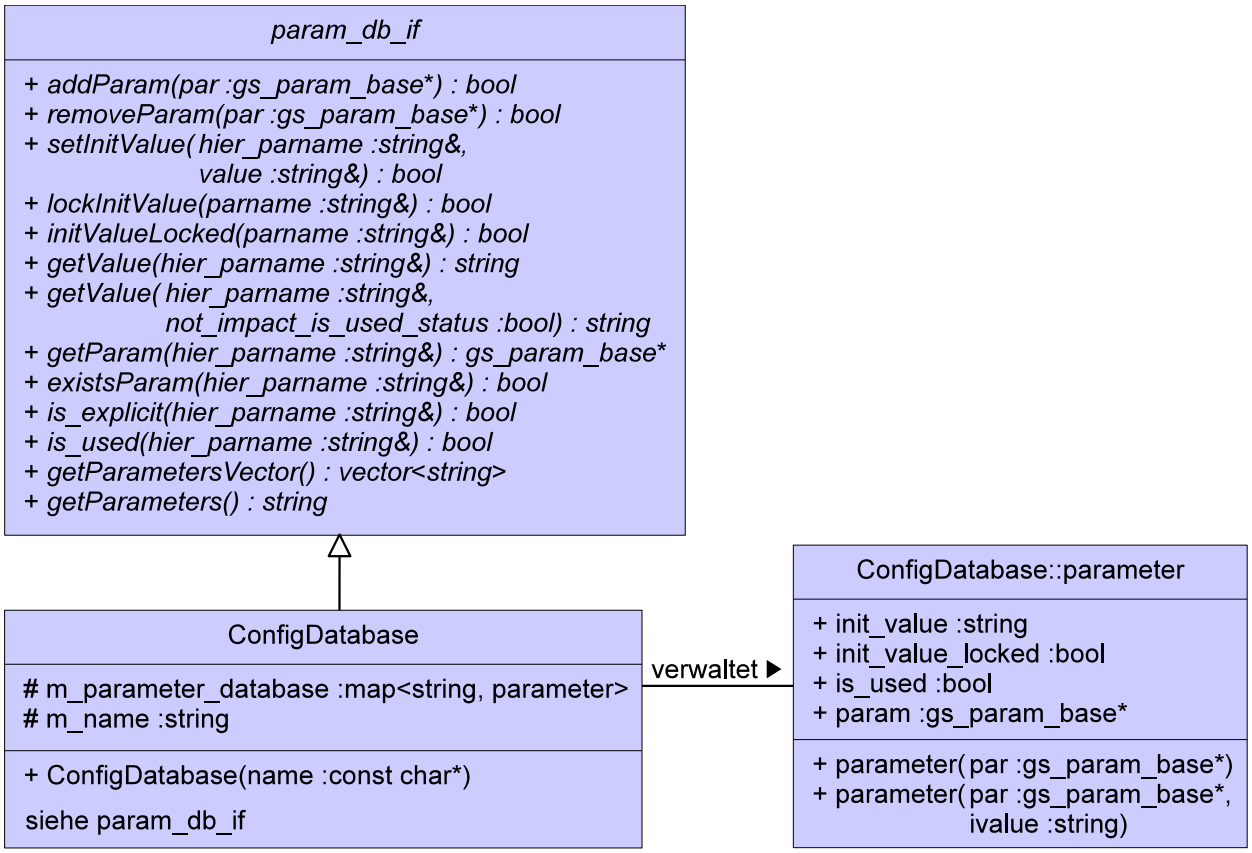


Abbildung 5.10.: Parameterdatenbank im Konfigurations-Plug-in (Klassendiagramm)

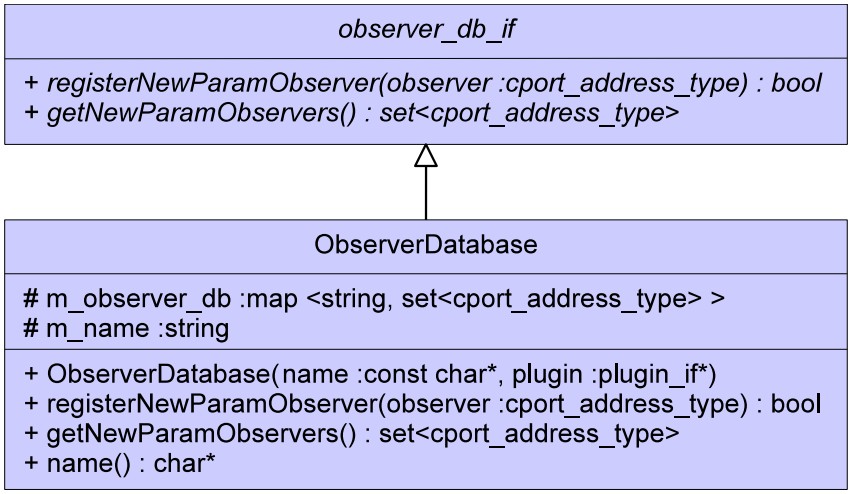


Abbildung 5.11.: Observerdatenbank im Konfigurations-Plug-in (Klassendiagramm)

Zu Debugzwecken liefert die Plug-in-Klasse den Namen „ConfigPlugin“ in der Funktion `name` zurück und implementiert das Interface `command_if` (vgl. Abbildung 4.5 auf Seite 74).

Aufbau Parameterdatenbank

Abbildung 5.10 stellt Details der nativen Default-Parameterdatenbank des Konfigurations-Plug-ins dar. Das Interface `param_db_if` deklariert die von der Klasse `ConfigDatabase` zu implementierenden Funktionen. Im Wesentlichen werden alle die Modell-Parameter betreffenden Anfragen an das Plug-in an die Datenbank weitergegeben. Die tatsächliche Speicherung der Modell-Parameter wird im Member `m_parameter_database` vorgenommen. In der aktuellen Implementierung ist die Datenbank als einfacher assoziativer Container realisiert. Eine Map der C++-Standard-Template-Library (STL) stellt die notwendige Funktionalität bereit. Für die Verbesserung der Performance oder Erweiterung mit optionalen Suchfähigkeiten oder Ähnlichem wäre es denkbar, die Implementierung gegen eine ausgereifte Datenbank auszutauschen, was hier aber nicht untersucht wird. Für die Speicherung der Parameterdaten dient der klasseninterne Hilfstyp `parameter`: Er speichert für implizite Parameter den Initialwert (als Zeichenkette), für explizite Parameter den GreenConfig-Parameter (als Pointer) und einige zusätzliche Status des Parameters.

Middleware-Kommandos

Der Konfigurationsservice verwendet die Aufzählung `ConfigCommand`. Deren Konstanten identifizieren die Kommandos, die in Middleware-Transaktionen des Konfigurations-Services übertragen werden. Die Bedeutungen werden in Tabelle 5.12 auf der nächsten Seite erläutert. Dort werden zwei mögliche Kommunikationsrichtungen unterschieden: Ein Kommando gehört entweder der Richtung User-API nach Plug-in an, oder es wird vom Plug-in an eine User-API gesendet. Die Rückgabe von abgefragten Werten ist jeweils auch auf dem Rückweg (Rückkehren des `transport`-Funktionsaufrufs) der Transaktion möglich. Welche Attribute der Transaktionen von einem Kommando verwendet werden, kann der Dokumentation [ScKl11a]^i entnommen werden.

Austauschbare Datenbank

Wird das Plug-in automatisch oder in seiner Standardkonfiguration erzeugt, wird automatisch die oben beschriebene Parameterdatenbank (Klasse `ConfigDatabase`) instanziiert und verwendet. Es ist aber möglich, eine andere Parameterdatenbank zu implementieren und dem Plug-in optional im Konstruktor zu übergeben, so lange sie das Interface `param_db_if` implementiert. Auf diese Weise kann eine Adapterdatenbank zu einem anderen Konfigurationsmechanismus und seiner Datenbank erstellt werden. In einem solchen Adapter können die GreenConfig-Parameter (und die Parameter aller Adapter-User-APIs) in den anderen Mechanismus gespiegelt werden, sowie die Modell-Parameter des anderen Mechanismus in

Kommando	**Bedeutung**
Kommunikationsrichtung: User-API → Konfigurations-Plug-in	
CMD_ADD_PARAM	registriert ein neues Parameterobjekt beim Plug-in
CMD_SET_INIT_VAL	setzt den Initialwert eines Parameters – erzeugt einen impliziten Parameter, wenn notwendig
CMD_LOCK_INIT_VAL	sperrt den Initialwert eines Parameters gegen erneutes Setzen
CMD_GET_VAL	liefert den Wert eines (impliziten oder expliziten) Parameters als Zeichenkette
CMD_GET_PARAM	liefert den Pointer auf einen expliziten Parameter
CMD_EXISTS_PARAM	liefert die Information, ob ein Parameter (implizit oder explizit) existiert
CMD_GET_PARAM_LIST_VEC	liefert eine Liste (aller oder spezifizierter) von Parameternamen von impliziten und expliziten Parametern
CMD_REGISTER_NEW_PARAM_OBSERVER	registriert einen neuen Callback an den angegebenen Beobachter für neue (implizite oder explizite) Parameter
CMD_REMOVE_PARAM	entfernt (beim Löschen) einen expliziten Parameter vom Plug-in
CMD_UNREGISTER_PARAM_CALLBACKS	löscht (z.B. beim Löschen eines Moduls) alle Callbacks (d.h. sowohl Callbacks für neue Parameter als auch alle Callbacks von Parameterobjekten) für den angegebenen Beobachter
CMD_PARAM_HAS_BEEN_ACCESSES	liefert die Information, ob ein Parameter bereits gelesen wurde
Kommunikationsrichtung: Konfigurations-Plug-in → User-API	
CMD_NOTIFY_NEW_PARAM_OBSERVER	benachrichtigt einen registrierten Beobachter über einen neuen (impliziten oder expliziten) Parameter

Tabelle 5.12.: Konfigurations-Middleware-Kommandos

GreenConfig-Parameter gespiegelt werden, die von der Adapterdatenbank instanziiert werden können.

Ein reales Beispiel für eine auf diese Weise ausgetauschte Datenbank ist der Adapter zum CoWare Platform-Architect, der im Abschnitt 6.1 auf Seite 132 beschrieben und im Beispiel B.6 nachvollzogen werden kann (Variante 4). Die austauschbare Datenbank ist eines der Integrationsverfahren, die in Abschnitt 5.9.3 auf Seite 122 vorgestellt werden.

5.6. GreenConfig-API

Die GreenConfig-API erlaubt globalen Zugriff auf alle Modell-Parameter im System. Sie kann sowohl innerhalb des Modells verwendet werden, beispielsweise von Parent-Modulen, die ihre Child-Module konfigurieren sollen, als auch von einem Konfigurations-Tool wie der Testbench oder einer IDE.

Aufbau

Abbildung 5.13 zeigt das Klassendiagramm der an der Tool-API beteiligten Klassen.

Zwei verschiedene GreenConfig-APIs implementieren das Interface `cnf_api_if`: Die Klasse `GCnf_Api` ist die im Normalfall verwendete GreenConfig-API, die Klasse `GCnf_private_Api` ist die im Abschnitt 5.7 näher erläuterte private GreenConfig-API. Für die Verbindung zur Middleware leitet die GreenConfig-API von drei bereits bekannten Middleware-Interfaces ab.

Das Interface `cnf_api_if` deklariert die rein virtuellen Zugriffsfunktionen, die von einer GreenConfig-API implementiert werden müssen. Zusätzlich definiert das Interface nicht-virtuelle Convenience-Template-Funktionen, die eine Typkonvertierung für den Abruf von Parameterwerten oder -objekten vornehmen. Des Weiteren enthält das Interface ein Makro für das Registrieren von Callbacks für Beobachter neuer Parameter.

Die Funktionen `addPar` und `removePar` werden von GreenConfig-Parameterobjekten beim Konstruieren und Zerstören aufgerufen, um die Datenbank zu aktualisieren. Die Funktionen `setInitValue` und `lockInitValue` setzen und sperren den Initialwert eines impliziten Parameters. Verschiedene get-Funktionen liefern Parameterwerte, -objekte oder -listen. Die Funktion `existsParam` liefert die Information, ob ein Parameter existiert (implizit oder explizit). `getPar` kann auch genutzt werden, um festzustellen, ob ein expliziter Parameter existiert, da sie das Parameterobjekt oder Null zurückgibt. Mit der Funktion `is_used` kann abgefragt werden, ob auf einen Parameter bereits ein wertneutraler Zugriff getätigt wurde. So können unbenutzte Parameter (z.B. entstanden durch Tippfehler in Konfigurationsdateien) identifiziert werden.

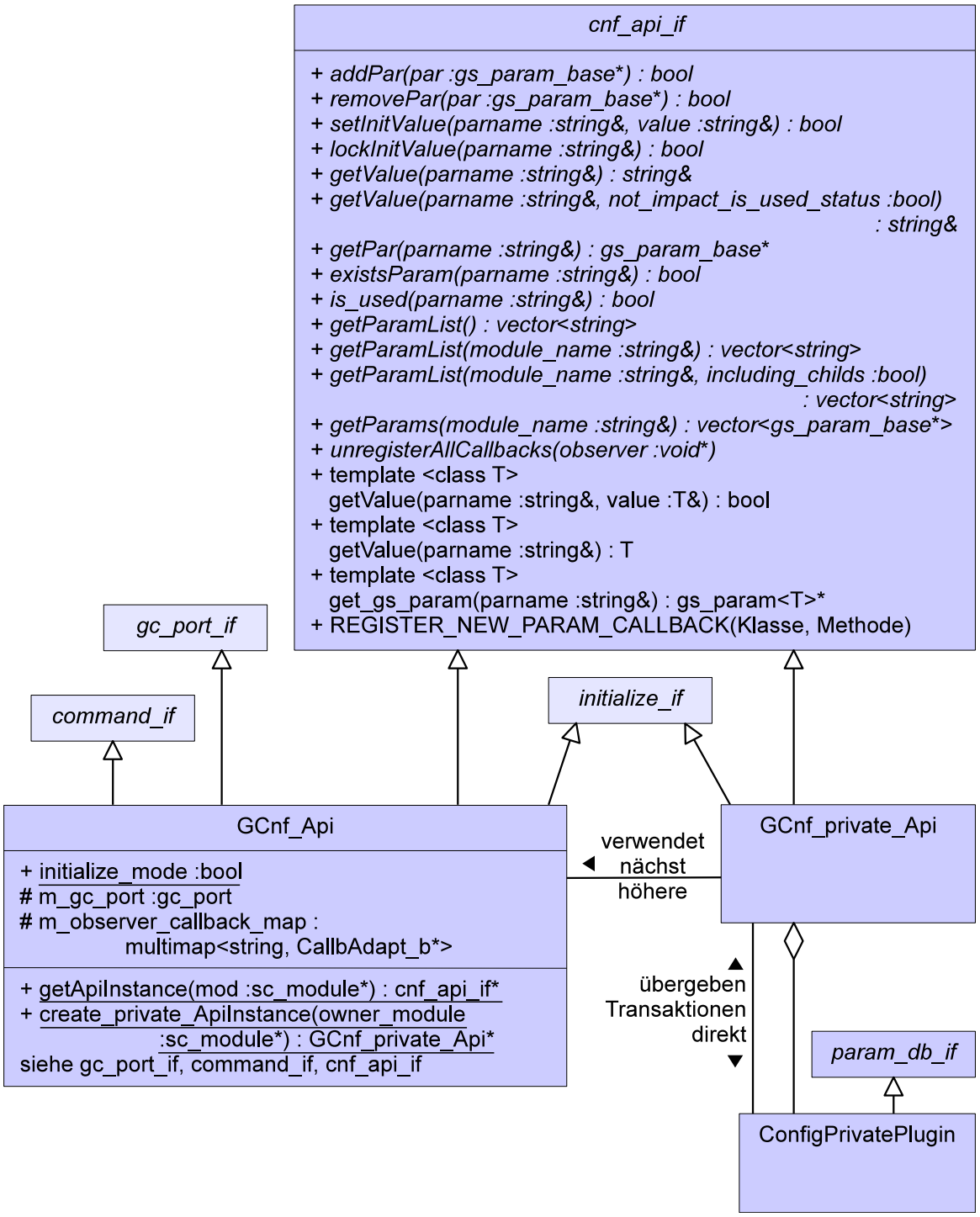


Abbildung 5.13.: GreenConfig-API (Klassendiagramm)

Callback für neue Parameter

Die Callbacks für Ereignisse unabhängig von Parameterobjekten aus Abschnitt 5.3 können für Methoden mit der folgenden Signatur registriert werden:

```
void config_callback(const std::string parname, const std::string value);
```

Das Makro `REGISTER_NEW_PARAM_CALLBACK` registriert einen solchen Callback, der die übergebene Methode aufruft, wenn ein Parameter neu erstellt wird (implizit oder explizit). Die Funktion `unregisterAllCallbacks` entfernt alle Callbacks an den übergebenen Beobachter.

Zugriff

Die Modell-Middleware und auch die GreenConfig-API unterstützen verschiedene Konzepte für die Existenz und den Zugriff auf User-APIs: Einerseits kann jedes Objekt, das eine User-API benötigt, sich eine Instanz erzeugen und mit ihr arbeiten. Andererseits ist es möglich, sich mit mehreren Objekten eine Instanz der GreenConfig-API zu teilen.

Der Zugriff auf GreenConfig-APIs ist über die statische Funktion `getApiInstance` der API-Klasse `GCnf_Api` geregelt. Sie bekommt eine Information über das anfragende Objekt und liefert die zuständige User-API zurück (vgl. Klassendiagramm in Abbildung 5.13).

Prinzipiell ist die GreenConfig-API für beide Konzepte geeignet – das Verhalten kann zentral im Quellcode dieser Funktion geändert werden. Ist es gewünscht, den Absender von Middleware-Transaktionen einem bestimmten Modul zuzuordnen, so ist es möglich, jedem Modul eine eigene User-API zu geben[12]. Da dies aus Benutzersicht zunächst jedoch nicht notwendig ist, gibt diese Funktion immer die gleiche GreenConfig-API-Instanz zurück.

Eine wesentliche Aufgabe dieser statischen Funktion ist zudem die Auswahl und Rückgabe privater GreenConfig-APIs, wenn es eine solche gibt, die für das anfragende Modul zuständig ist. Details dazu werden im folgenden Abschnitt 5.7 dargestellt.

GreenConfig-Parameter verwenden ebenfalls diese statische Funktion, sodass sie automatisch die für ihr Parent-Modul zuständige GreenConfig-API zurückgegeben bekommen. Ihr Parent-Modul ist ihnen bekannt, da sie – wie in Abschnitt 5.4.3 beschrieben – SystemC-Objekte sind.

5.7. Private GreenConfig-API

GreenConfig unterstützt das Verstecken von Modell-Parametern in Submodulen (vgl. Anforderung KA18) mit Hilfe der privaten GreenConfig-API `GCnf_private_Api`.

Wenn ein Modul private Parameter enthalten soll, die nur dort und in Child-Modulen sichtbar sein sollen, kann es eine private GreenConfig-API erzeugen. Mit dem im vorherigen

[12] Implementierungsdetails wie kleinere Container (`m_observer_callback_map`) oder Caching-Effekte könnten Performance-Gründe für diesen Ansatz darstellen.

Abschnitt beschriebenen Zugriffsmechanismus mit der statischen Funktion wird sichergestellt, dass sowohl die Modell-Parameter als auch die Nutzer von GreenConfig-APIs die zuständige private API verwenden.

Abbildung 5.13 auf Seite 116 stellt den Zusammenhang privater Config-APIs mit dem Rest des Systems sowie deren groben Aufbau dar. Auffällig ist, dass die private API keine direkte Verbindung zur Middleware hat (d.h. sie leitet weder vom Port-Interface ab noch besitzt sie einen Port). Stattdessen ruft sie die hierarchisch nächst höhere GreenConfig-API-Instanz über die statische Zugriffsfunktion ab. Folglich bietet die private API Zugriff auf alle darüber liegenden Modell-Parameter, verbirgt allerdings die internen, nicht explizit als öffentlich markierten Parameter vor dem darüber liegenden System.

Die private GreenConfig-API soll aus Sicht des Benutzers weitgehend die gleichen Aufgaben wie die normale GreenConfig-API in Zusammenspiel mit dem Konfigurations-Plug-in bereitstellen. Deswegen ähnelt ihre Implementierung der GreenConfig-API, und sie besitzt ein privates Plug-in (Klasse `ConfigPrivatePlugin`), dessen Implementierung der des Konfigurations-Plug-ins gleicht. Das erleichtert die Wartbarkeit und macht Erweiterungen einfacher. Die Informationen gelangen nicht nach außen, da die private API und ihr privates Plug-in nicht über die Middleware kommunizieren, sondern die Transaktionen einander direkt übergeben.

Abbildung 5.14 zeigt ein Beispiel mit privaten und nicht privaten GreenConfig-APIs. Die dunkel umrahmten Kästen in Parametern und Modulen (ModF) verdeutlichen jeweils, welche private API verwendet wird, also von der statischen Funktion `getApiInstance` zurückgegeben wird. Es wird auch deutlich, dass private APIs geschachtelt werden können (ModB erzeugt eine eigene, die diejenige von ModA verwendet).

Die Datenbank im Config_Plugin enthält neben den Parametern des Moduls Mod0 nur die Modell-Parameter des Moduls ModA und die von dessen Child-Modul, die von ModA bei seiner privaten API explizit als öffentlich angegeben wurden[13]. Die private API von ModA ermöglicht nur Zugriff auf Parameter, die in der private API ModA.ModB explizit als öffentlich angegeben wurden.

Sicherheit privater GreenConfig-APIs

In Gesprächen mit Industrievertretern in der CCI-Arbeitsgruppe ist der Wunsch einiger Firmen nach einem möglichst sicheren Verbergen von Parametern deutlich geworden. In diesen Fällen soll ein binär ausgeliefertes Modell dem Kunden möglichst wenig Einblick in den internen Aufbau gewähren. Ungewollt sichtbare Parameter würden ein Sicherheitsproblem darstellen.

Private GreenConfig-APIs bieten nur eingeschränkte Sicherheit gegen böswillige Angriffe:

- Es wird gewährleistet, dass die in GreenConfig spezifizierten Schnittstellen keine Informationen über versteckte Parameter preisgeben. Das bedeutet, dass ohne Manipulation

[13]Für das Tool sind nur die in der Datenbank des Konfigurations-Plug-ins enthaltenen Parameter sichtbar.

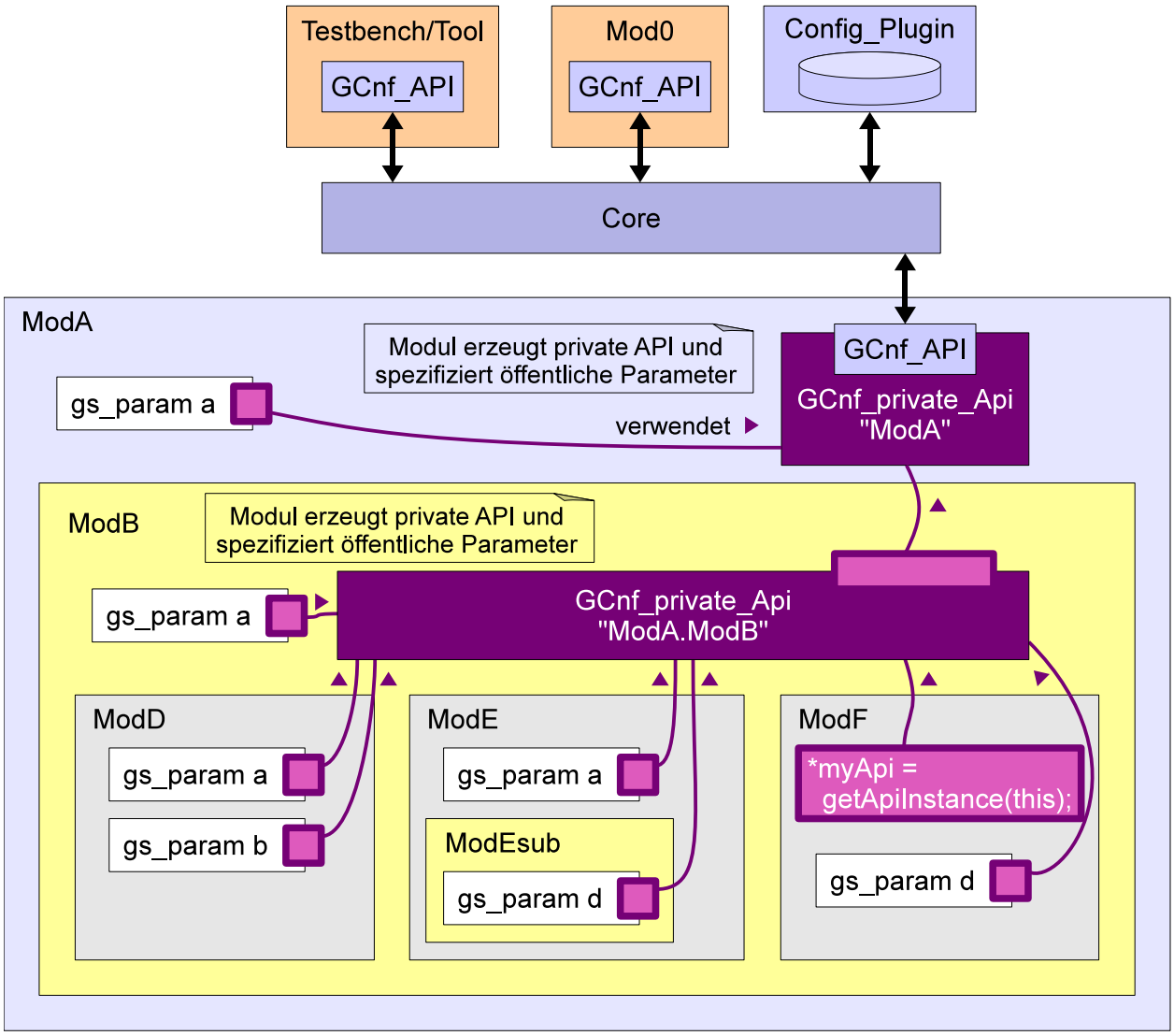


Abbildung 5.14.: Beispiel mit privaten GreenConfig-APIs

der Quellen des Konfigurationsmechanismus Modelle und Tools keinen Zugriff auf versteckte Parameter bekommen.

Die folgenden Schwachstellen sollen exemplarisch einen Überblick über die Art der Manipulationen bzw. des Missbrauchs geben, die Angriffe auf die Sicherheit ermöglichen:

1. *Direkter Zugriff*: Liegt ein Subsystem, das versteckte Parameter enthalten soll, im Quelltext vor (Header oder sogar Implementierung), sind vielfältige Manipulationen und Zugriffe denkbar[14]. Unter diesen Bedingungen ist echte Sicherheit allerdings generell nicht erreichbar.

2. *SystemC-Objekthierarchie durchsuchen*: Da die GreenConfig-Parameter SystemC-Objekte sind (vgl. Abschnitt 5.4.3), kann ein Angreifer die SystemC-Schnittstellen verwenden, d.h. die SystemC-Objekthierarchie durchsuchen, um direkt auf Parameterobjekte zuzugreifen[15].

3. `getApiInstance` *mit fremdem Modulpointer aufrufen*: Die statische `getApiInstance`-Funktion kann von einem Angreifer mit einem falschen Modulpointer aufgerufen werden. Wenn dieser Pointer ein Modul ist, das Zugriff auf eine private API hat, wird diese auch an den Angreifer zurückgegeben. Fremde Modulpointer können beispielsweise direkt selbst instanziierte sein[16] oder (wie unter vorangehendem Punkt beschrieben) in der SystemC-Objekthierarchie gefunden werden.

4. *Konfigurationsmechanismus manipulieren*: Die Manipulation des Konfigurationsmechanismus selbst ermöglicht naheliegende Angriffe: Die Funktion `getApiInstance` kann manipuliert werden, immer eine nicht-private API zurückzugeben, sodass sich auch die eigentlich versteckten Parameter bei dieser anmelden und somit global erreichbar sind. Ein anderer Ansatzpunkt für Manipulationen sind die Parameterobjekte selbst.

Warum sind die beschriebenen Sicherheitsprobleme trotzdem hinnehmbar? Zum einen ist der primäre Anwendungsfall, dem Anwender interne Konfigurationsfähigkeiten zu verbergen, um ihn nicht zu verwirren und vor sinnlosen Änderungen der Konfiguration zu bewahren. Zum anderen setzen die oben aufgezählten Angriffe eine „böse“ Absicht des Angreifers voraus. Angriffe mit diesem Hintergrund sind generell sehr schwer zu vermeiden, da eine vollständige Verschleierung der internen Strukturen notwendig ist. Bei SystemC-Modellen liegt häufig der Quellcode vor, was Verschleierung quasi unmöglich macht. Ist das nicht der Fall, gibt es Methoden des Reverse-Engineering.

Soll ein Modell gegen solche Angriffe geschützt werden, sollte es keinen bekannten Konfigurationsmechanismus verwenden und nur binär weitergegeben werden.

[14] Abhilfe schafft hier die Binärauslieferung von Modellen, wie sie von einigen Industrieunternehmen vorgenommen wird (vgl. [OSCI09][i]).

[15] GreenConfig-Parameter sind SystemC-Objekte, damit der automatische Benennungsmechanismus feststellen kann, welchem Modul sie angehören.

[16] Ein nur binär ausgeliefertes, also nicht im Quellcode vorliegendes, Subsystem könnte den direkten Zugriff auf den Pointer eines Moduls verhindern, indem es die private API erst in einem zusätzlichen Hierarchielevel (Submodul) versteckt.

5.8. Weitere Tool-APIs

GreenConfig stellt einige weitere Tool-APIs zur Verfügung[17], deren Details und Verwendung der Dokumentation [ScKl11a][i] entnommen werden können. Alle im Folgenden gelisteten Tool-APIs verwenden intern eine GreenConfig-API.

- *Konfigurationsdatei-Tool*
  Das Konfigurationsdatei-Tool in der Klasse `ConfigFile_Tool` liest Konfigurationsdateien ein und wendet sie an. Diese Dateien haben eine einfache Syntax. Jede Zeile definiert einen Parameterwert, beginnend mit dem Parameternamen. Mit einem Leerzeichen getrennt schließt sich die Zeichenkettenrepräsentation des Parameterwerts an.

- *Lua-Konfigurationsdatei-Tool*
  Ein weiteres Konfigurationsdatei-Tool wird von der Klasse `LuaFile_Tool` bereitgestellt. Es liest Konfigurationsdateien mit Lua-Syntax (eine Skriptprogrammiersprache [Ieru06] [Pont10][i]) ein und wendet sie an.

- *Kommandozeilen-Parser-Tool*
  Die Klasse `CommandLineConfigParser` stellt einen Parser zur Verfügung, der aus den übergebenen Kommandozeilenargumenten Optionen für die Konfiguration einzelner Parameter extrahiert und anwendet.

- *Kommandozeilen-Tool*
  Eine besondere Tool-API ist die Klasse `CommandLine_Tool`. Sie fügt der Simulation eine Kommandozeile hinzu, mit der auf verschiedene Konfigurationseigenschaften zugegriffen werden kann.

5.9. Integration und Adapter-APIs

Dieser Abschnitt fasst in einer Übersicht zusammen, wie der GreenConfig-Konfigurationsmechanismus die Integration anderer Konfigurationsmechanismen ermöglicht. Dabei setzt er die in anderen Kapiteln behandelten Grundlagen und jeweils besonders geeigneten Anwendungsbeispiele in Beziehung und zeigt den Zusammenhang zu GreenConfig auf. Auf diesen Ansätzen basieren die im Kapitel 6 beschriebenen Adapter-APIs, auf die in diesem Abschnitt mehrfach verwiesen wird.

5.9.1. Konfigurationsansätze integrieren

Im Abschnitt 3.1.2 wurden zwei Konfigurationsansätze vorgestellt: Konfigurations-Interfaces und Class-Wrapper. Im Vergleich der beiden Konfigurationsansätze wurde festgestellt, dass

[17] Es gibt neben den GreenConfig-Parametern keine weitere native Modell-API, da gleichwertige Alternativen nicht sinnvoll wären. Adapter-Modell-APIs werden in Abschnitt 5.9 beschrieben.

der Class-Wrapper-Ansatz mächtiger ist und deswegen im GreenConfig-Konfigurationsmechanismus primär verwendet wird. Im Abschnitt 3.3.3 wurde dann allgemein beschrieben, wie ein proprietärer Konfigurationsmechanismus, der Konfigurations-Interfaces verwendet, in Class-Wrapper integriert werden kann (für die beiden Richtungen PIU und UIP). Da die GreenConfig-Parameter als Repräsentanten des Class-Wrapper-Ansatzes die dort genannten Voraussetzungen[18] erfüllen, ist eine Integration von Konfigurations-Interfaces in beiden Richtungen möglich.

Die Umsetzung einer solchen Integration ist in Abschnitt 6.3 für das ARM CASI-Interface beschrieben.

Damit ist gezeigt, dass der Konfigurationsmechanismus GreenConfig das Hauptziel dieser Arbeit für unterschiedliche Konfigurationsansätze erfüllen kann.

5.9.2. Konfigurationsrangfolge

Im Abschnitt 3.1.3 wurden die zeitliche und die hierarchische Konfigurationsrangfolge eingeführt und untersucht. GreenConfig verwendet als Resultat die zeitliche Konfigurationsrangfolge. Im Abschnitt 3.3.4 wurden allgemeine Möglichkeiten der PIU-Integration vorgestellt, deren Anwendbarkeit sich in den zur Verfügung stehenden Merkmalen unterscheidet. GreenConfig ermöglicht sämtliche vorgestellten Integrationsansätze.

Eine Realisierung des Ansatzes mit Verwendung von Initialwertsperren wird ausführlich im Abschnitt 6.4 vorgestellt. Das OVM-Anwendungsbeispiel im Abschnitt 6.5 realisiert den Ansatz mit Verwendung des Originals.

Damit ist gezeigt, dass der Konfigurationsmechanismus GreenConfig das Hauptziel dieser Arbeit für unterschiedliche Konfigurationsrangfolgen erfüllen kann.

5.9.3. Integrationsrichtungen

Für die Konfigurations-Interoperabilität mit proprietären Tools bzw. IDEs ist ein Abgleich der Modell-Parameter zwischen dem betreffenden proprietären Konfigurationsmechanismus und GreenConfig notwendig. Abschnitt 3.3.1 hat bereits generelle Ansätze für die Integrationsrichtungen PIU und UIP diskutiert. Dieser Abschnitt beschreibt einige mit GreenConfig realisierbare Ansätze für die notwendigen Adapter für beide Richtungen.

Abbildung 5.15 auf der gegenüberliegenden Seite zeigt eine Übersicht über die möglichen Positionen von Adaptern. Die verschiedenen Szenarien werden im Verlauf des Abschnitts erläutert. Allgemein zeigt die Abbildung ein konfigurierbares Real-Modell (verschiedene IP-Blöcke, die unter Verwendung verschiedener Konfigurationsmechanismen entwickelt wurden). Zusätzlich ist im unteren Bereich die Modell-Middleware mit dem Konfigurations-Plug-in enthalten, die von einigen User-APIs verwendet wird. Schließlich ist auf der linken Seite ein proprietäres Tool dargestellt, das eine eigene proprietäre Konfigurations-API bereitstellt und eine eigene Parameterdatenbank verwaltet.

[18] Konkret sind die Voraussetzungen durch die Callbacks gegeben.

Die verschiedenen Adapter-Szenarien sind in der Abbildung dunkel mit heller Schrift dargestellt und sollen in den folgenden Unterabschnitten einzeln aufgegriffen werden.

Technische Details zu Umsetzungen aller Integrationen werden in Kapitel 6 in den Anwendungsbeispielen präsentiert. Dort wird auch deutlich, dass die Anforderung AA6, die proprietären Modelle möglichst nicht modifizieren zu müssen, erfüllt wird. Damit ist gezeigt, dass der Konfigurationsmechanismus GreenConfig das Hauptziel dieser Arbeit für die Integration anderer Mechanismen erfüllt.

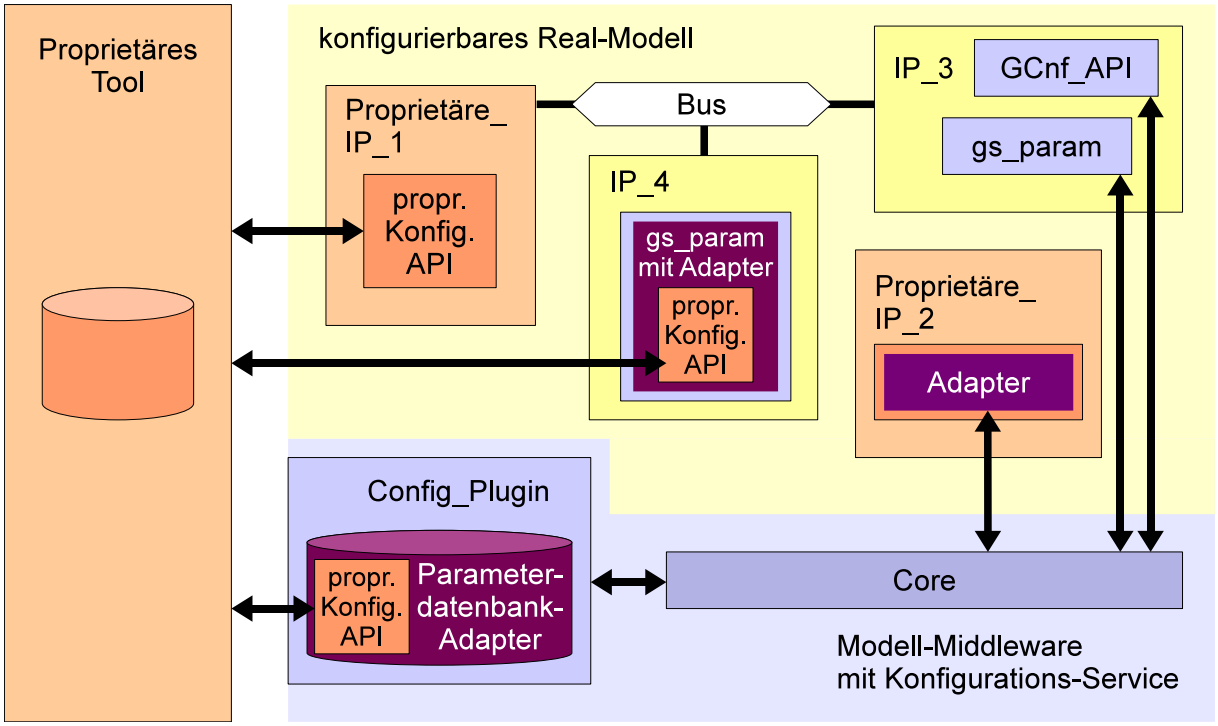


Abbildung 5.15.: Adapter-Übersicht; Adapter sind dunkel mit heller Schrift dargestellt

Integration mit Datenbankadapter

Das Modul IP_3 ist ein mit GreenConfig entwickeltes Modul und verwendet den universellen Konfigurationsmechanismus. Soll für diesen Anwendungsfall die UIP-Integration erreicht werden, ist ein Parameterdatenbank-Adapter eine Möglichkeit.

Das proprietäre Modul IP_1 verwendet den originalen proprietären Konfigurationsmechanismus und ist somit zunächst für GreenConfig unsichtbar. Soll für diesen Anwendungsfall die PIU-Integration erreicht werden, kann ebenfalls ein Parameterdatenbank-Adapter verwendet werden.

Da jedes andere Modul zwangsläufig entweder an GreenConfig oder den proprietären Mechanismus angeschlossen ist, kann es folglich über die gleichen Datenbank-Adapteransätze integriert werden.

Für die beiden in Abschnitt 3.3.1 geforderten Integrationsrichtungen PIU und UIP ist das Konfigurations-Plug-in durch seinen Aufbau (vgl. „Austauschbare Datenbank“ in Abschnitt

5.5) vorbereitet und ermöglicht verschiedene Integrationsstufen. Von den Stufen kann je nach Mächtigkeit und Aufbau der Ziel-Entwicklungsumgebung eine Stufe gewählt werden.

1. Die vollständige Ersetzung des Konfigurations-Plug-ins und der dort enthaltenen Datenbank kann erfolgen, wenn ausschließlich die Datenbank der Ziel-IDE verwendet werden soll. Das bedeutet, dass die verfügbaren Merkmale – auch für die universellen Modelle – von den Merkmalen des proprietären Mechanismus bestimmt bzw. beschränkt werden. Es ist unwahrscheinlich, dass diese kompromisslose Methode bei einem existierenden Mechanismus zur Anwendung kommen kann[19].

2. Eine für die praktische Anwendung relevante Abstufung ist, das Konfigurations-Plug-in zu erhalten und die Datenbank unter Verwendung der vorgesehenen Schnittstellen durch eine leicht modifizierte auszutauschen. Damit können die von dem Zielmechanismus nicht unterstützten Merkmale trotzdem bereitgestellt werden. Die modifizierte Datenbank spiegelt dann den Inhalt der proprietären Datenbank. Diese Realisierung kann in Abbildung 5.15 im Parameterdatenbank-Adapter erfolgen.

3. Eine mit der vorherigen verwandte Variante ist, das Konfigurations-Plug-in beizubehalten und die an das Plug-in angeschlossene Datenbank komplett durch einen Adapter zum Zielmechanismus zu ersetzen. Das setzt allerdings voraus, dass die Ziel-Datenbank die vom Plug-in benötigten Funktionen unterstützt, beispielsweise Pointer auf die Modell-Parameter speichern kann. Auch dieser Ansatz ist in Abbildung 5.15 im Parameterdatenbank-Adapter möglich.

Ein reales Beispiel für einen Datenbankadapter ist in Abschnitt 6.1 (UIP-Integration) für SCML gezeigt.

Proprietäres Modell integrieren

Für die Einbindung eines proprietären Modells in den universellen Mechanismus (PIU-Integration) kann dessen Konfigurationsverhalten mit einem Adapter in ein bezüglich der Konfiguration universelles Modell gewandelt werden. Das Modul Proprietäre_IP_2 in Abbildung 5.15 ist ein solches Modell. Der Adapter imitiert eine proprietäre Schnittstelle zum Modell und übersetzt sie zu GreenConfig.

Die Voraussetzung ist, dass GreenConfig sämtliche relevanten Merkmale des proprietären Mechanismus unterstützt. Ansonsten muss GreenConfig zunächst um die fehlenden Merkmale erweitert werden. Irrelevante Merkmale können alternativ im Adapter ignoriert oder beispielsweise mit einer Fehlermeldung behandelt werden. Durch die Wahl des mächtigeren Class-Wrapper-Ansatzes als native Modell-API für GreenConfig wird die Möglichkeit der Realisierung von Adaptern sichergestellt.

Soll sogar ein Modell eingebunden werden, das für einen Konfigurationsmechanismus eines dritten Herstellers entworfen wurde, ist über den Umweg des universellen Mechanismus

[19]Diese Integrationsvariante ist in Abbildung 5.15 nicht dargestellt.

auch die Integration zwischen unterschiedlichen proprietären Konfigurationsmechanismen möglich.

Reale Beispiele für diese Integration sind in Abschnitt 6.1 (PIU-Integration) für SCML und in Abschnitt 6.2 für CCSS-Parameter gezeigt.

Universelles Modell integrieren

Ist die Integration mit einem Datenbankadapter nicht möglich oder nicht gewünscht, kann ein universelles Modell auch direkt in einen proprietären Konfigurationsmechanismus eingebunden werden. In Abbildung 5.15 ist dieser Ansatz für IP_4 dargestellt. Ein Parameter-Adapter imitiert die universelle GreenConfig-Parameter-Schnittstelle (`gs_param`) und übersetzt sie direkt in Aufrufe an die gewünschte proprietäre Konfigurations-API. Dieser Ansatz beschränkt je nach verfügbaren Merkmalen des proprietären Mechanismus den Funktionsumfang der universellen Parameter erheblich.

Ein reales Beispiel für diese Integration ist in Abschnitt 6.2 (UIP-Integration) für den Innovator gezeigt.

5.10. Modell-Untersuchung mit GreenConfig

Die Modell-Konfiguration ist nach Abschnitt 2.3 das Beeinflussen von Eigenschaften eines Modells, was von einem Konfigurationsmechanismus folglich geleistet werden muss. Dort wird aber zusätzlich die Modell-Untersuchung eingeführt, was die Eigenschaften eines Modells auszulesen bedeutet. Die vorangehenden Abschnitte dieses Kapitels zeigen, dass der GreenConfig-Konfigurationsmechanismus als Nebenprodukt der Modell-Konfiguration auch die Modell-Untersuchung für Parameter ermöglicht: Sowohl Initialwerte als auch Werte expliziter Parameter können ausgelesen werden. Parameterobjekte erlauben mit den Callbacks sogar detailliertes Beobachten der Aktivitäten und Zustände. Einerseits sind diese Informationen bereits für die Modell-Konfiguration interessant, beispielsweise um sich gegenseitig beeinflussende oder voneinander abhängende Modell-Parameter zu realisieren. Andererseits können diese Fähigkeiten für sehr viel weitergehende Zwecke verwendet werden, beispielsweise für eine umfangreiche Analyse der Modellaktivitäten durch externe Tools und das Debuggen durch den Nutzer.

Da bei der Entwicklung von GreenConfig stets Wert auf möglichst hohe Performance gelegt wurde (vgl. auch Abschnitt 6.9), können GreenConfig-Parameter auch für die Analyse, beispielsweise während der Simulationsphase, verwendet werden.

5.11. Analyse-Service

Als Folge der Eignung von GreenConfig für die Modell-Untersuchung dienen die GreenConfig-Parameter als Informationsquelle für einen Analyse-Service.

Der Analyse-Service ist abhängig vom Konfigurations-Service, da er dessen Fähigkeiten zur Modell-Untersuchung nutzt. Zusätzlich ist der Analyse-Service eine Demonstration der Möglichkeiten der Modell-Middleware, auch für andere Zwecke als die der Konfiguration sinnvoll genutzt zu werden. Der Analyse-Service trägt einen Beitrag zur Konfigurations-Interoperabilität bei: er erfüllt die Anforderung KA5 (siehe Seite 44), Konfigurationen in verschiedene Dateien ausgeben zu können. Der Analyse-Service trägt zudem in einigen Aspekten zur Meta-Interoperabilität bei, indem er beispielsweise Ausgabeformate unterstützt, die von anderen Tools dargestellt werden können. Da der Analyse-Service nicht vorwiegend der Konfigurations-Interoperabilität dient, sind Details im Anhang A zu finden.

6. Anwendungsbeispiele und Tests

Inhalt

Die in diesem Kapitel präsentierten Anwendungsbeispiele sind einerseits Demonstrationen der Fähigkeiten der in dieser Arbeit vorgestellten Konzepte und Software und beinhalten andererseits Tests.

Darüber hinausgehende Einzeltests, sowohl systematische Regressions-Tests von besonders wichtigen oder komplexen Funktionen als auch kleine und größere Demonstrationsbeispiele, sind jeweils in den entsprechenden Projektquellen zu finden (z.B. [Schr11, Schr10a][i] und [Schr11b]).

Zunächst wird gezeigt, wie die Integrationen in die kommerziellen Werkzeuge CoWare Platform-Architect (Abschnitt 6.1), Synopsys Innovator (Abschnitt 6.2) und ARM CASI (Abschnitt 6.3) realisiert sind. Abschnitt 6.4 beschreibt die Nachbildung hierarchischer Rangfolge mit GreenConfig. Den umgekehrten Weg beschreitet der OVM-Adapter (Abschnitt 6.5), der die hierarchische Rangfolge in die zeitliche Rangfolge von GreenConfig integriert.

Im Abschnitt 6.6 wird ein großes Integrationsbeispiel vorgestellt, das die zuvor eingeführten Adapter in einer Simulation vereinigt und diese in einer kommerziellen Entwicklungsumgebung ausführt.

Abschnitt 6.7 behandelt SCRSI, ein modulares Programm für SystemC-Entwicklungstools. Es beinhaltet Services, die eine Benutzeroberfläche für GreenControl-Services bereitstellen und diese für zusätzliche Erweiterungen nutzen.

Im Abschnitt 6.8.1 wird demonstriert, wie ein bestehendes Register-Framework mit GreenConfig-Parametern effizient und einfach mit den Konfigurations-Fähigkeiten dieses Mechanismus ausgestattet werden kann.

Die vorgestellten Anwendungsbeispiele sind – mit Ausnahme von GreenReg – Proof-of-Concept-Implementierungen. Die in diesem Kapitel referenzierten Beispiele sind im Anhang B zusammengestellt, wo auf deren Quelle verwiesen wird.

6.1. CoWare Platform-Architect

Dieser Abschnitt beschreibt beide Integrationsverfahren des GreenConfig-Konfigurationsmechanismus (vgl. PIU- und UIP-Integration aus Abschnitt 3.3) in die kommerzielle Entwicklungsumgebung CoWare Platform-Architect (jetzt Synopsys). Wie bereits im Review in Abschnitt 2.6.3 festgestellt wurde, verwendet der CoWare-Konfigurationsmechanismus den Class-Wrapper-Ansatz. Die Parameter-Objekte aus der CoWare-Bibliothek werden SCML-Properties genannt (`scml_property`). Die im Folgenden vorgestellten Adapter sind Umsetzungen der in Abschnitt 3.3.2 beschriebenen Varianten.

PIU-Integration: CoWare-Modell in GreenConfig-Entwicklungsumgebung

Ein mit SCML-Properties konfigurierbares Modell soll in der universellen GreenConfig-Umgebung konfiguriert werden (PIU-Integration). Das Ziel der Integration ist, das SCML-Property-Objekt so zu modifizieren oder zu ersetzen, dass das Modell ohne Änderung am Quellcode weiterhin wie mit den Originalobjekten arbeiten und genauso konfiguriert werden kann.

In diesem Fall ist der Wunsch nach einem unveränderten Quellcode erfüllbar, da die SCML-Bibliothek mit dem Include scml.h eingebunden wird, das von GreenConfig ersetzt werden kann. Im GreenConfig-Projekt existiert eine Proof-of-Concept-Implementierung: Die Datei scml.h in [Schr11]^{i1} kann dem Compiler statt der originalen Datei angegeben werden, wodurch ein unmodifizierter Modell-Quellcode mit seinem Header-Include bestehen bleibt und trotzdem der Adapter verwendet wird. Alternativ kann die SCML-Adapter-Datei[2] direkt im Modell-Quellcode eingebunden werden. Falls die Ersatzklasse parallel zu der originalen existieren soll, kann dies durch Definieren des Makros `ENABLE_SCML_NAMESPACE` ermöglicht werden, was die modifizierte `scml_property`-Klasse in den Namespace `gs::cnf` platziert.

Die Ersatzklasse heißt ebenso wie das Original `scml_property`, und es existieren Spezialisierungen für die vom Original unterstützen Datentypen (`int`, `unsigned int`, `bool`, `double`, `string`). Abbildung 6.1 zeigt das Klassendiagramm. Die Klasse enthält sämtliche der SCML-Dokumentation entnommenen Funktionen des Originals. Der Adapter übersetzt sämtliche Funktionsaufrufe in Aufrufe auf eine interne Tool-API (Klasse `Scml_Api`). Diese API greift auf GreenConfig-Parameter zu, die über die herkömmliche GreenConfig-API (`GCnf_Api`) erzeugt und verwaltet werden. Die Umsetzung dieses Adapters ist aufgrund der Ähnlichkeiten

[1] SCML-Header-Datei: greencontrol/gcnf/apis/scmlApi/scml.h in [Schr11]^{i}
[2] SCML-Adapter-Datei: greencontrol/gcnf/apis/scmlApi/scml_property.h in [Schr11]^{i}

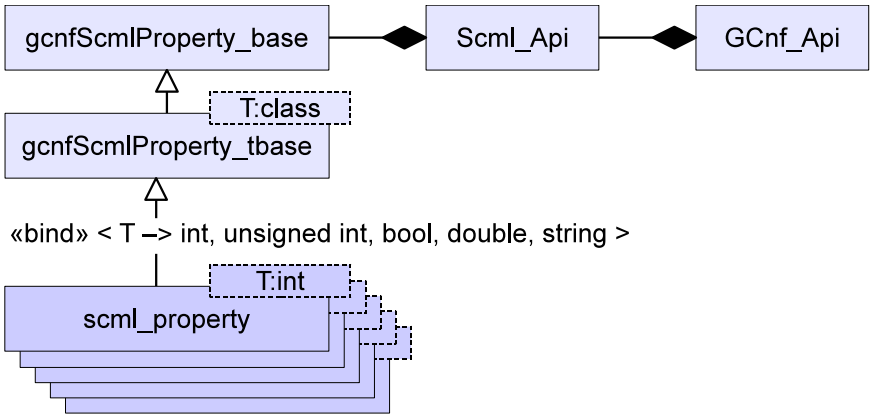


Abbildung 6.1.: User-API-Adapter für SCML-Property (Klassendiagramm)

der Mechanismen relativ geradlinig – beide Mechanismen verwenden Class-Wrapper-Objekte mit ähnlicher Funktionsweise.

Zwei Beispielsysteme sollen die erfolgreiche Integration demonstrieren:

Das Beispiel B.4 im Anhang besteht unter anderem aus einem nach den Regeln der SCML-Dokumentation entworfenen Modul (Scml_Orig_IP). Es besitzt `scml_property`s und kann darüber konfiguriert werden[3]. Dieses Beispiel ist sowohl im Platform-Creator[4] als auch in einer GreenConfig-Simulation lauffähig und kann jeweils wie ein natives Modell konfiguriert werden. Zu beachten ist, dass die Modell-Parameter des CoWare-Modells auch in der Umgebung der GreenConfig-Simulation nicht die Beschränkungen der SCML-Properties aufheben dürfen, damit sich die Semantik und das Verhalten der Parameter im Modell nicht ändert.

Das Beispiel B.5 ist ein mit dem SCWizard[5] erstelltes Modell. Die automatisch auch als Konstruktor-Parameter angelegten SCML-Properties können ebenso problemlos mit dem Adapter, durch den sie ersetzt werden, konfiguriert werden.

Beide Beispiele können ohne jede Quellcode-Änderung in eine GreenConfig-Simulation außerhalb der CoWare-IDE übernommen werden. Sie müssen lediglich mit jeweils einem anderen Include-Pfad für die Datei scml.h neu kompiliert werden. Das Setzen der Parameter ist mit den GreenConfig-Mechanismen möglich (z.B. C++-Tool-API, Konfigurationsdatei, GreenScript usw.), das Verhalten aus Sicht des Modells ist identisch mit dem in der CoWare-Simulation: Der Initialwert wird, wenn er gesetzt ist, während der Konstruktion des Parameters angewendet und überschreibt einen möglichen Default-Konstruktor-Wert.

[3]Die in dem vorgestellten Beispiel eingesetzten SCML-Properties sind *nicht* gleichzeitig Konstruktor-Parameter, was der Standard für vom CoWare-Assistenten (SCWizard) erzeugte Module ist. Diese Vorgehensweise widerspricht der in dieser Arbeit propagierten Natur der Konfiguration, da Konstruktor-Parameter eine Form der manuellen Konfiguration sind, die durch Konfigurationsmechanismen ersetzt werden soll. In der CoWare-IDE (Platform-Creator) werden die Modell-Parameter dieser Modelle folgerichtig sowohl als Konstruktor- als auch als Konfigurations-Parameter angezeigt und sind zweifach konfigurierbar.
[4]Der Platform-Creator ist das Simulationstool des Platform-Architect.
[5]Der SCWizard ist ein Assistent innerhalb des Platform-Architects zum Erstellen von Modellen.

UIP-Integration: Universelles GreenConfig-Modell im CoWare Platform-Architect

Ein universelles Modell verwendet die GreenConfig-Modell-API in Form von GreenConfig-Parametern. Das Einbinden und Konfigurieren dieses universellen Modells in die CoWare-IDE (den Platform-Creator) erfordert die UIP-Integration. Die Integration kann auf verschiedene Weise realisiert werden. Im Folgenden werden hierfür vier Varianten diskutiert, deren Umfang und Robustheit mit jeder Variante zunimmt:

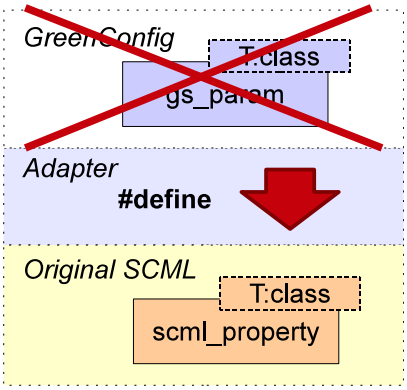
Alle vier Varianten können mit dem Beispiel B.6 im Platform-Creator nachvollzogen werden, indem das Makro `COWARE_VARIANTE` in der Datei globals.h mit den Werten `1` bis `4` definiert wird. Die Testbench-Datei ist im Platform-Creator überflüssig, die Initialisierungen für Variante 4 müssen dort mit der sogenannten „Main Configuration“ vorgenommen werden. Für die Umsetzung der Varianten ist vor allem die Datei gs_to_scml_wrapper.h relevant. Abbildung 6.2 auf der gegenüberliegenden Seite zeigt eine Übersicht über die vier Varianten.

Variante 1 ist einfach aber unzureichend: per `define` werden alle universellen Parameter (`gs_param`) in SCML-Properties (`scml_property`) umdefiniert. In Abbildung 6.2(a) ist das durch den großen Pfeil dargestellt. Das Verfahren funktioniert stellenweise für einige Merkmale und Datentypen, da die Syntax der beiden Mechanismen in Teilen gleich ist. Allerdings führen alle in SCML nicht möglichen Zugriffe auf die Parameter zu Compiler-Fehlern. Zudem hat ein GreenConfig-Konfigurator keinen Zugriff auf die Parameter. Gegenüber den nachfolgenden Varianten hat diese den Vorteil, dass die ersetzten Parameter originale SCML-Properties sind, die auch einer Typkontrolle standhalten würden, allerdings keinem Quellcode-Parsing vor dem Präprozessor.

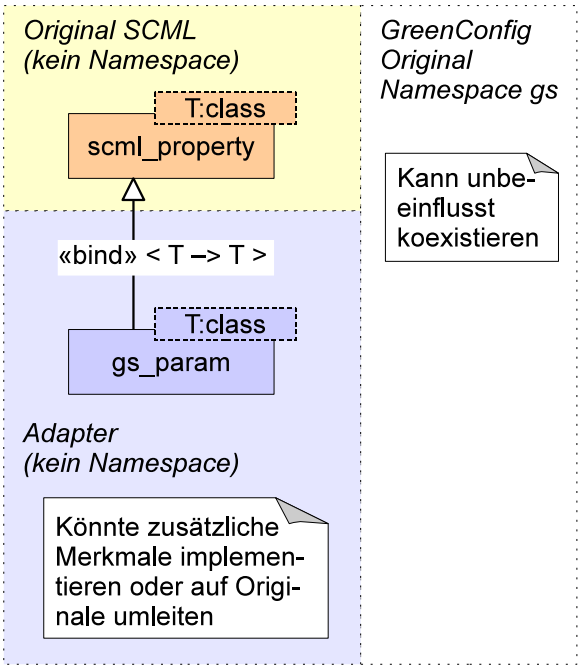
Die folgenden Varianten 2 bis 4 sind dagegen nur unter der Voraussetzung anwendbar, dass das Hinzufügen von Properties im Platform-Creator-GUI nicht an Kontrollen gebunden ist[6]. In der verwendeten Programmversion (vgl. Fußnote [7] auf Seite 18) war das Hinzufügen der modifizierten Parameter problemlos möglich.

Variante 2 ersetzt die im Modell verwendeten GreenConfig-Parameter durch neu erstellte Adapter gleichen Namens[7]. Diese Variante kann mit GreenConfig koexistieren, die Adapter müssen sich dann jedoch in einem anderen (oder keinem) Namespace befinden, siehe Abbildung 6.2(b). Im Modell muss dann dafür gesorgt werden, dass die neuen Adapter verwendet werden[8]. Das schränkt den Modell-Code geringfügig ein, da die GreenConfig-Parameter dort nicht in ihrem originalen Namespace (`gs::gs_param`), sondern ohne Namespace (`gs_param`) verwendet werden dürfen. Wie bei Variante 1 hat ein GreenConfig-Konfigurator keinen Zugriff auf die Parameter, da es sich nicht mehr um GreenConfig-Parameter handelt. Optional könnte der Adapter auch eine Verbindung zu GreenConfig herstellen[9], sodass letztere Einschränkung entfallen könnte. Dafür müsste der Adapter alle Merkmale, die SCML-Properties

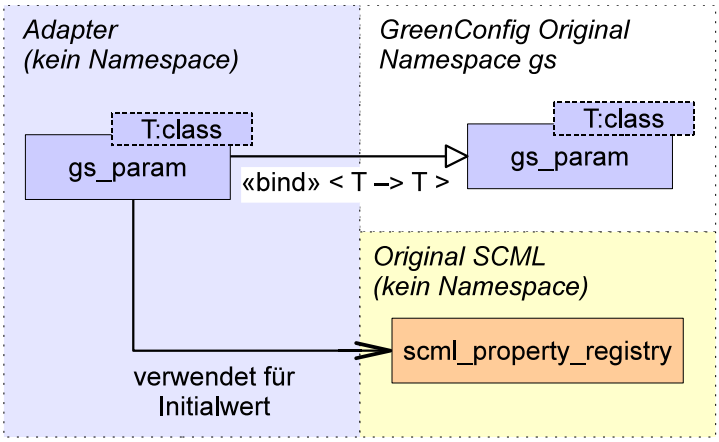
[6] Properties werden dem Platform-Creator vom Benutzer grafisch oder per Skript bekanntgemacht.
[7] Gemeint ist das Ersetzen der Parameter-Klasse, nicht die Modifikation des Modell-Quellcodes.
[8] Beispielsweise kann wie im Beispiel `using namespace gs` entfernt werden.
[9] Da es sinnvollere Varianten gibt, wurde dieser Zusatzaufwand für das Beispiel nicht umgesetzt.



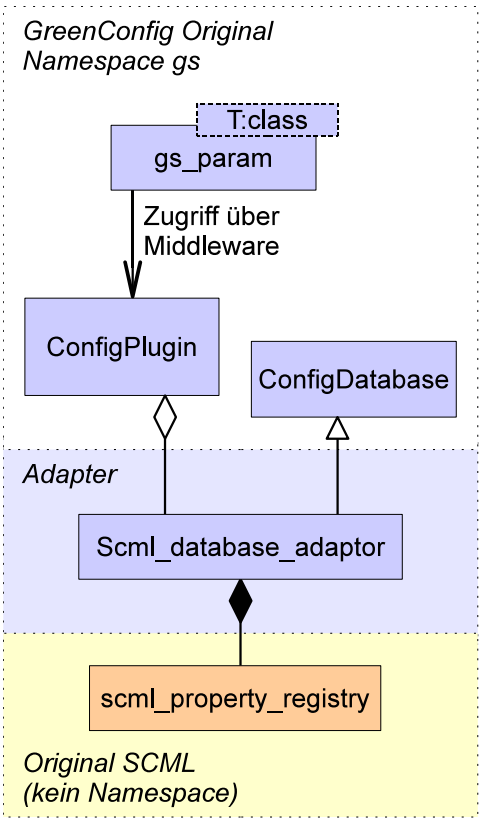
(a) Variante 1



(b) Variante 2



(c) Variante 3



(d) Variante 4

Abbildung 6.2.: Varianten der UIP-Integration im Platform-Architect

nicht zur Verfügung stellen, implementieren beziehungsweise auf originale GreenConfig-Parameter umleiten. Die folgende Variante 3 eignet sich hierfür jedoch besser.

Variante 3 ähnelt Variante 2 und ersetzt ebenfalls die im Modell verwendeten Parameter durch neu erstellte Adapter gleichen Namens ohne Namespace. Diese Adapter verwenden indes nicht die Fähigkeiten der SCML-Properties, sondern leiten von den originalen GreenConfig-Parametern ab mit dem Vorteil, deren Merkmale vollständig zu erben. Die Verbindung zum SCML-Mechanismus wird hergestellt, indem die SCML-Datenbank beim Konstruieren der Adapter direkt auf einen Initialwert für den entsprechenden Parameter abgefragt wird (z.B. mit dem Aufruf von `scml_property_registry::inst().getIntProperty()`). Diese Variante ist in Abbildung 6.2(c) dargestellt. Sie hat den Vorteil, dass ein GreenConfig-Konfigurator unverändert alle Merkmale des universellen Mechanismus verwenden kann, also nicht vom fremden Mechanismus eingeschränkt wird. Ebenso wie die zuvor beschriebenen Adapter müssen die Parameter im Modell ohne Namespace verwendet werden. Zudem wird im Adapter eine nicht dokumentierte Funktion (`hasProperty`) des SCML-Mechanismus verwendet.

Variante 4 ist eine Realisierung des in Abschnitt 5.5 auf Seite 113 beschriebenen Austauschs der Datenbank. Sie ist in Abbildung 6.2(d) dargestellt. Die vom Modell verwendeten Parameter bleiben unveränderte GreenConfig-Parameter. Die Verbindung zum SCML-Mechanismus wird auf ähnliche Weise wie in Variante 3 hergestellt, allerdings in einem Datenbank-Adapter im Service-Plug-in anstatt in Parameter-Adaptern. Der Datenbank-Adapter (Klasse `gs::cnf::Scml_database_adaptor` in der Datei scml_database_adaptor.h) fragt beim Erzeugen von GreenConfig-Parametern und bei der Abfrage von impliziten Parametern bei der SCML-Datenbank mit dem gleichen Befehl wie in Variante 3 einen potentiellen Initialwert ab. Ansonsten erbt der Datenbank-Adapter die Fähigkeiten der originalen GreenConfig-Datenbank. Diese Variante ist eine elegante Lösung, die die optimale Koexistenz zwischen beiden Konfigurationsmechanismen ermöglicht.

Das Beispiel B.5 kann für alle Varianten simuliert werden und zeigt in allen Fällen die gleichen Ergebnisse.

Fazit

Dieser Abschnitt zeigt, wie die PIU-Integration für einen proprietären Mechanismus, der den Class-Wrapper-Ansatz verwendet, realisiert werden kann. Proprietäre Modelle, die CoWare-SCML-Properties verwenden, können ohne Quellcode-Manipulation in einer universellen Simulation konfiguriert werden.

Die verschiedenen Varianten der UIP-Integration für das CoWare-Beispiel zeigen, dass potentiell vielfältige Möglichkeiten für die Integration bestehen. Trotzdem kann nicht allgemeingültig garantiert werden, dass ein universelles Modell in jeden proprietären Mechanismus eingebunden werden kann – was in dieser Arbeit auch nicht zwingend gefordert ist.

6.2. Synopsys Innovator

Eine weitere Entwicklungsumgebung, an der die Integrationsfähigkeiten vom GreenConfig-Framework (vgl. Abschnitt 3.3) demonstriert werden sollen, ist der im Abschnitt 2.6.5 eingeführte Innovator von Synopsys [Syno08][i10] mit den CCSS-Parametern nach Class-Wrapper-Ansatz.

Hier wird mit der weniger wichtigen UIP-Integration begonnen, da dort Grundlagen für die im Anschluss beschriebene PIU-Integration gelegt werden.

UIP-Integration: Universelles Modell im Innovator

Im Folgenden werden Parameter-Wrapper beschrieben, die es erlauben, GreenConfig-Parameter in einem mit Innovator simulierten Modell zu verwenden. Die Adapter werden vom Innovator wie eigene CCSS-Parameter erkannt und angezeigt und können dort konfiguriert werden. Der Quellcode ist im Beispiel B.7 zu finden.

In Versuchen hat sich herausgestellt, dass der Innovator-Quellcodeparser nicht so restriktiv ist, wie es die Vorgaben in den Hilfetexten vermuten lassen. Die im Folgenden beschriebene Integration funktioniert möglicherweise nur mit der getesteten Innovator-Version, da sie zum Teil auf unspezifiziertem und undokumentiertem Verhalten des Innovators beruht.

Zusammenfassend werden die GreenConfig-Parameter mit einer einfachen `using`-Anweisung durch neue Parameter-Wrapper ersetzt. Für den Benutzer sind die Wrapper weiterhin wie GreenConfig-Parameter zu verwenden, gleichzeitig sehen sie für den Innovator wie die proprietären CCSS-Parameter aus und werden ebenso behandelt und angezeigt. Das funktioniert, indem der Wrapper sowohl von GreenConfig-Parametern als auch von CCSS-Parametern ableitet und somit beide Eigenschaften erbt.

Die *Realisierung* erfolgt im Detail folgendermaßen:

Damit der Modell-Quellcode GreenConfig-Parameter (`gs::gs_param`) verwenden kann, wird eine Wrapper-Klasse `gs_ccss::gs_param` erstellt, die sich exakt wie ein GreenConfig-Parameter verwenden lässt. Zu diesem Zweck leitet die Wrapper-Klasse von der GreenConfig-Parameter-Klasse ab. Der Modell-Quellcode muss diesen Parameter-Wrapper anstatt der originalen GreenConfig-Parameter verwenden. Die einzigen Änderungen an der bestehenden SystemC-Anwendung, die GreenConfig-Parameter verwendet, sind einmal eine `using`-Anweisung für die neue Parameter-Wrapper-Klasse (`using gs_ccss::gs_param`), die statt der normal verwendeten zur Anwendung kommen muss und eine neue Angabe im Header. Die Folge daraus und eine Einschränkung sind, dass die Parameter im Modell-Quellcode ohne Namespace-Angabe verwendet werden müssen.

Damit der Innovator-Quellcode-Parser den Wrapper-Parameter als CCSS-Parameter erkennt, muss die Wrapper-Klasse zusätzlich von der CCSS-Parameterklasse `ccss_param` ableiten. Dieser Umstand ist hinreichend, damit der Innovator-Quellcode-Parser die Objekte als Parameter erkennt. Allerdings gelten die Einschränkungen der CCSS-Parameter bezüg-

[10] Es wurde die Innovator Version 2008.12-SP2, Build 20081202009 verwendet.

lich der Instanziierungsregeln und dem Setzen der Initialwerte auch für die neuen Parameter (siehe untenstehende Auflistung).

Eine Herausforderung stellt der Umstand dar, dass CCSS-Parameter mit einem Makro erzeugt werden, GreenConfig-Parameter aber nicht. Die CCSS-Parameter-Makros erzeugen für jeden Parameter eine eigene Klasse, die statisch den Parameternamen verwaltet. Diese Klasse wird dem originalen CCSS-Parameter als Template-Argument übergeben, sodass der Parameter über diese Klasse Zugriff auf seinen Namen hat[11]. Dieses Verfahren ist für die Parameter-Wrapper nicht realisierbar, da sie nicht mit einem Makro erstellt werden und somit keine Klasse generiert werden kann, die sich auf den Parameternamen bezieht (d.h. den Parameternamen im Klassennamen hat). Es waren keine Stellen außer im Konstruktor des `ccss_param`-Objekts auffindbar, an denen dieser Parametername aus der Klasse abgerufen wird. Dort wird der Name u.a. verwendet, um das `sc_object` mit diesem Namen zu konstruieren, von dem wiederum der `ccss_param` ableitet. Unter der Annahme, dass die Nutzung dieser Klasse tatsächlich so eingeschränkt ist, wird es möglich, eine gemeinsame Klasse für sämtliche Parameter-Wrapper zu verwenden, die dann beim Ableiten jedem CCSS-Parameter als Template-Parameter übergeben wird. Der Parametername muss dann auf den Namen des gerade zu erzeugenden Parameters aktualisiert werden, bevor der `ccss_param`-Konstruktor aufgerufen wird, der auf den Namen zugreift. Um das Setzen des Namens vor dem Konstruktoraufruf des `ccss_param`s durchführen zu können, wird der Parameter-Wrapper von einer weiteren Hilfsklasse (`base_class_for_set_name_call`) abgeleitet, die keine andere Aufgabe hat, als in ihrem Konstruktor den Aufruf der statischen Funktion zum Setzen des Namens zu ermöglichen. Durch die richtige Ableitungsreihenfolge und Konstruktoraufrufreihenfolge in der Initialisierungsliste des Wrapper-Parameters kann der Parametername gesetzt werden, bevor der CCSS-Parameter konstruiert wird.

Da der `ccss_param` sowie auch der originale `gs_param` von `sc_object` ableiten, muss diese Ableitung virtuell geschehen. Die Klasse `sc_object` ist also eine virtuelle Basisklasse von dem Wrapper sowie seiner beiden Parameter-Basisklassen. Dadurch ist es notwendig, dass der `sc_object`-Konstruktor auch vom Wrapper-Konstruktor aufgerufen wird, dort dann mit dem richtigen Namen. Da virtuelle Basisklassen-Konstruktoren *vor* den anderen (nicht-virtuellen) aufgerufen werden, muss auch die Hilfsklasse zum Setzen des Namens eine virtuelle Basisklasse sein, damit sie vor dem `sc_object`-Konstruktor aufgerufen werden kann.

Abbildung 6.3 zeigt die wichtigsten Klassen des Ableitungsbaums des Parameter-Wrappers `gs_ccss::gs_param`.

Die Übernahme von Initialisierungswerten aus dem Innovator-GUI in die Simulation wird im Wrapper-Konstruktor mit dem dafür vorgesehenen Funktionsaufruf (`conditional_init`) durchgeführt.

Damit sämtliche Zugriffe auf die Basisklassen des Wrappers zuverlässig den richtigen (aktuellen) Wert verwenden (vor allem ist dies notwendig für die CCSS-Parameter-Basisklas-

[11] Sollte diese Klasse tatsächlich nur diesem Zweck dienen, ist sie überflüssig und könnte durch einen einfachen Konstruktor-Parameter ersetzt werden. Allerdings ist diese Optimierung Synopsys Aufgabe und möglicherweise wird diese spezielle Klasse an weiteren Stellen benötigt, die nicht offen zugänglich sind.

se, die sich der Kontrolle des Adapters entzieht), werden die Werte von der `gs_param`- zur `ccss_param`-Basisklasse synchronisiert. Zu diesem Zweck registriert der Wrapper eine interne Callback-Funktion auf dem Basis-GreenConfig-Parameter und gibt jede Änderung sofort an den Basis-CCSS-Parameter weiter. Lesende Zugriffe werden vom CCSS-Parameter gelesen.

Einige `using`-Anweisungen erlauben den Zugriff auf CCSS-Parameterfunktionen.

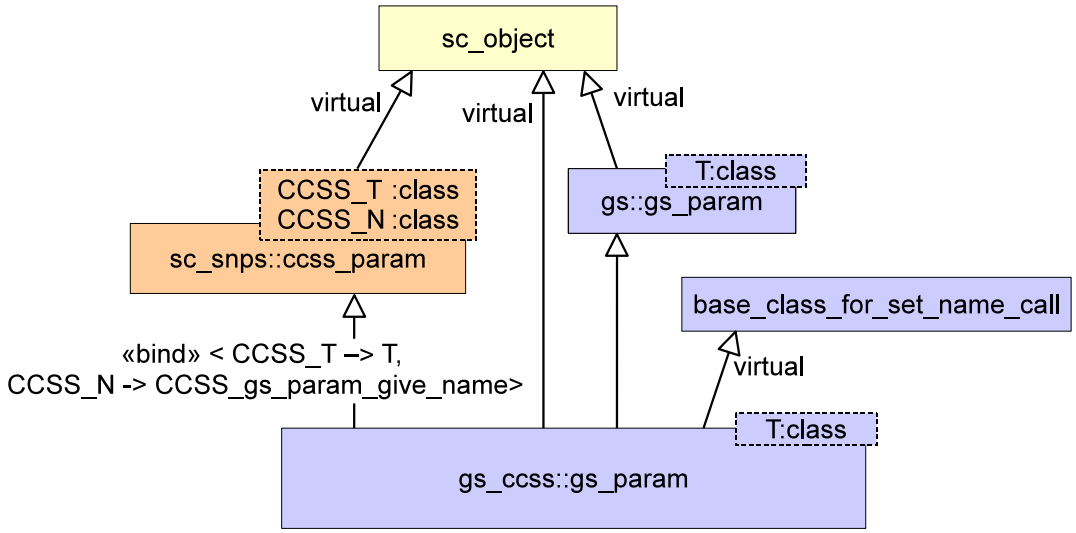


Abbildung 6.3.: Parameter-Wrapper `gs_ccss::gs_param` (Klassendiagramm)

Zusammengefasst bestehen die folgenden Einschränkungen bzw. Modifikationen bei der Verwendung der GreenConfig-Parameter-Wrapper im Innovator im Gegensatz zur gewohnten Verwendung der originalen GreenConfig-Parameter:

- Im Header muss die Parameter-Wrapper-Datei (anstatt GreenConfig) eingebunden werden.
- Das using statement `using gs_ccss::gs_param` muss verwendet werden.
- Der lokale Konstruktor-Parametername muss dem Variablennamen entsprechen, sonst funktionieren die Anzeige und das Setzen eines Initialwertes im Innovator nicht (im Innovator wird der Variablenname verwendet, in der laufenden Simulation der Konstruktorname).
- Der Innovator erkennt nur Parameter, die als Klassenmember definiert sind.
- In der Proof-of-Concept-Implementierung sind nur zwei ausgewählte Konstruktoren verfügbar, prinzipiell sind alle anderen GreenConfig-Parameter-Konstruktoren auch möglich.

PIU-Integration: Innovator-Modell in universeller GreenConfig-Umgebung

Mit dem Innovator erzeugte Modelle, die CCSS-Parameter für die Konfiguration verwenden, können in einer GreenConfig-Simulation außerhalb der Innovator-IDE konfiguriert und simuliert werden. Die Adapter sind im Beispiel B.8 zu finden (die Klasse `ccss_param` bzw. das

Makro `CCSS_PARAMETER(P_TYPE, P_NAME)`). Diese ersetzen transparent CCSS-Parameter und realisieren sie intern als GreenConfig-Parameter.

Zunächst wird die CCSS-Header-Datei durch eine modifizierte ersetzt. Dadurch kann der Modell-Quellcode ohne Modifikation verwendet werden und mit den modifizierten Daten neu kompiliert werden.

In der Ersatzdatei werden die Wrapper definiert. Das Makro zum Erstellen eines CCSS-Parameters wird ersetzt durch eines, das den Parameter-Wrapper und eine Hilfsklasse instanziiert, ähnlich der im vorigen Abschnitt für die statische Verwaltung des Parameternamens eingeführten Klasse. Der CCSS-Parameter-Wrapper definiert und implementiert die gleichen Funktionen, die auch der originale CCSS-Parameter zur Verfügung stellt. Dadurch sieht er für das Modell wie ein CCSS-Parameter aus und lässt sich genau so verwenden. Tatsächlich aber leitet der Wrapper von der GreenConfig-Parameterklasse `gs_param` ab und bildet die Funktionsaufrufe auf den Basis-Parameter ab.

Die vom Makro zusätzlich erzeugte Hilfsklasse verwaltet den Parameternamen, der nicht beim Konstruieren des Parameters übergeben werden kann. Der Initialwert, der vor dem Instanziieren des Parameters potentiell in der Datenbank existiert, wird wieder von einer virtuellen Basisklasse verwaltet. Dieser Initialwert wird während der Konstruktion aus der Datenbank ausgelesen – und zwar bevor sich der Parameter dort registriert – und wird verwendet und verglichen, wenn die Funktion `conditional_init` aufgerufen wird.

Fazit

Der Synopsys Innovator ist eine durch die Marktposition des Herstellers sehr wichtige und verbreitete Entwicklungsumgebung. Die Integration in ein bedeutsames Tool dieser Art ist ein Erfolg. Es ist ein Vertreter der Programme, die Quellcode-Parsing für die Erkennung von Konfigurations-Parametern und deren Werte verwenden und somit eine wichtige Studie, um die Vielseitigkeit des GreenConfig-Frameworks zu zeigen. Innovator-Modelle können mit dem universellen Mechanismus konfiguriert werden, und GreenConfig-Modelle können im Innovator konfiguriert werden.

6.3. ARM CASI

Dieser Abschnitt beschreibt beide Integrationsverfahren des GreenConfig-Mechanismus in Entwicklungsumgebungen, die das Cycle Accurate Simulation Interface (CASI) verwenden (vgl. Review in Abschnitt 2.6.4).

Der Konfigurationsmechanismus im ARM CASI ist ein Vertreter des Konfigurations-Interface-Ansatzes. Die im Folgenden vorgestellten Adapter sind folglich Umsetzungen der im Abschnitt 3.3.3 beschriebenen Varianten.

PIU-Integration: CASI-Modell in GreenConfig-Umgebung

Der CASI-Konfigurationsmechanismus fordert von einem konfigurierbarem Modell, dass es von der CASI-Klasse `casi_module` ableitet. Für diese Klasse ist einerseits eine rudimentäre Implementierung in der Bibliothek vorgegeben, andererseits dürfen ihre Funktionen auch vom Benutzermodul überschrieben werden. Aus diesem Grund existieren Proof-of-Concept-Implementierungen für die beiden Adapter-Varianten (e) und (f) (vgl. Abschnitt 3.3.3), die mit beiden Optionen umgehen können. Da die Adapter-Variante (f) mit einem systemweiten Monitor, der die proprietäre CASI-API verwendet, Einschränkungen bezüglich der Funktionalität aufweist, sollte sie nicht als alleinige Lösung verwendet werden. Wenn der Benutzer das Konfigurations-Interface nicht selbst implementiert, kann die Adapter-Variante (e) eingesetzt werden, um eine bessere Kopplung zu gewährleisten. Der Adapter (f) ist auf eine gemeinsame Nutzung mit Adapter (e) verwendende Modelle vorbereitet[12].

Beispiel B.9 ist ein CASI-Modell, das in einer herstellerfremden OSCI-Simulation eingebettet ist und von der Testbench konfiguriert wird. Es wendet die Adapter-Variante (e) an, das bedeutet, es verwendet eine CASI-Bibliothek[13] mit einer neuen Implementierung der Klasse `casi_module`, bei der die Speicherung der Parameterwerte in Form von `gs_param<string>` anstatt `string` erfolgt.

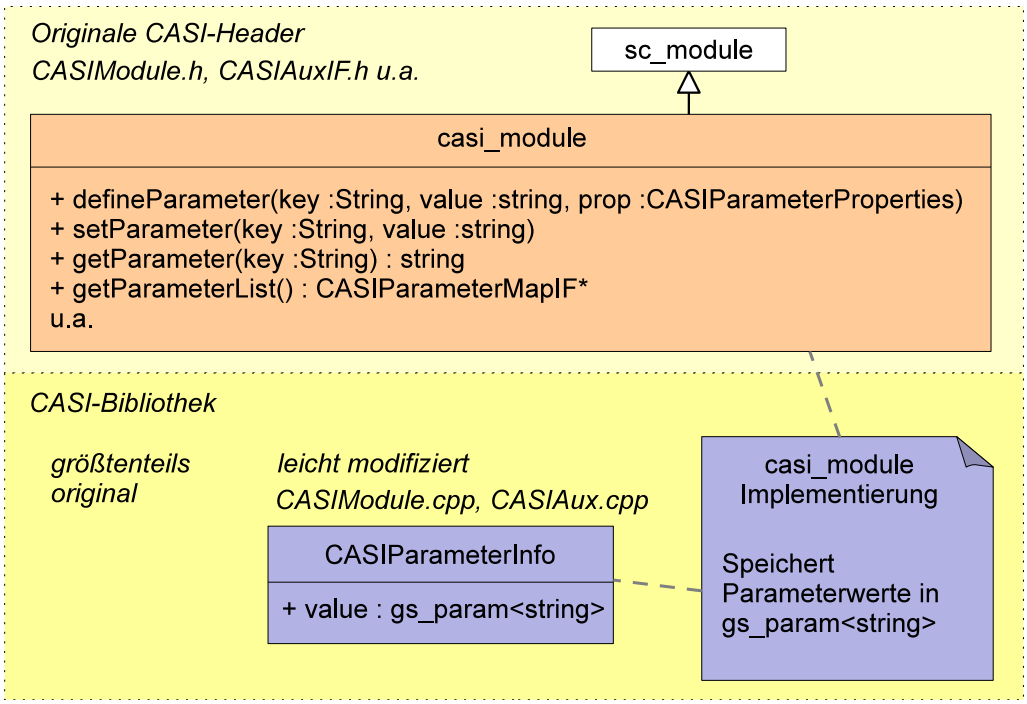


Abbildung 6.4.: PIU-CASI-Adapter für CASI-Parameter in einer GreenConfig-Umgebung (nach Adapter-Variante (e))

Für diesen Adapter können die originalen CASI-Quellen verwendet werden, die zu einer leicht modifizierten Bibliothek kompiliert werden, gegen die das Modell gelinkt wird (vgl.

[12]Vorgriff: Der Adapter legt bereits gespiegelte Parameter nicht erneut als Schattenparameter an.
[13]In diesem Beispiel wird die CASI-Version 1.1.0 verwendet.

Abbildung 6.4). Die folgende Modifikation ist notwendig: Die Original-Implementierung der Klasse `casi_module` verwendet eine Variable vom Typ `CASIParameterInfo`, in der für die Speicherung des Parameterwertes eine Zeichenkette `string` verwendet wird. Dieser (ausschließlich in der Implementierung intern definierte und verwendete[14]) Datentyp wird für den Adapter geändert und verwendet statt der einfachen Zeichenkette einen GreenConfig-Parameter vom Typ `gs_param<string>`. Zu diesem Zweck wird das GreenControl-Projekt in der Implementierung eingebunden (Datei CASIAux.cpp). Die ebenfalls implementierungsinterne Klasse `CASIParameterMap` wird so angepasst, dass keine temporären Parameter erstellt werden und keine Parameter kopiert werden müssen. Um einen GreenConfig-Initialwert wie vorgesehen zu übernehmen, wird ein solcher Wert in der Funktion, die einen Parameter definiert (Datei CASIModule.cpp), abgefragt und verwendet.

Das Ergebnis ist eine modifizierte Bibliothek, die das CASI-Interface implementiert und allein durch neues Linken jeden Modell-Parameter eines CASI-Modells in einem universellen GreenConfig-Parameter speichert und folglich jeder Zugriff auf den Parameterwert vom GreenConfig-Parameter registriert und weiterverarbeitet werden kann. Solange die Funktionen der modifizierten Bibliothek verwendet werden – und nicht vom Modell überschrieben werden – wird eine optimale Umsetzung zwischen den verschiedenen Konfigurationsansätzen erreicht.

Der Aufbau des Beispiels ist wie folgt: Die Testbench bindet das GreenConfig-Framework, ein originales CASI-Modul und ein GreenConfig-Beobachtungsmodul ein. Das CASI-Modul (mit einem Untermodul) verwendet das CASI-Konfigurations-Interface, sodass es unmodifiziert auch in einer CASI-Umgebung funktioniert. Das Beobachtungsmodul `Gs_Observer_IP` hat über die GreenConfig-API uneingeschränkten[15] Zugriff auf die mit dem Adapter umgesetzten CASI-Modell-Parameter. Es protokolliert alle Modell-Parameterzugriffe. Die Testbench verwendet die GreenConfig-API, um Initialwerte der Modell-Parameter des CASI-Moduls zu setzen, instanziiert das Modul anschließend und verwendet das proprietäre Konfigurations-Interface des Moduls. Die Terminalausgaben protokollieren das nahtlose Zusammenspiel des proprietären Interfaces mit dem GreenConfig-Interface.

In einem weiteren Beispiel B.10 wird die Adapter-Variante (f) demonstriert. Auch in diesem Beispiel ist ein CASI-Modul in einer OSCI-Simulation eingebettet und wird von der Testbench mit GreenConfig konfiguriert. Das Beispiel bildet ein CASI-Modul nach, das durch Überschreiben des Konfigurations-Interfaces eine eigene Implementierung für den Konfigura-

[14] Implementierungsintern bedeutet hier, dass alle notwendigen Definitionen und verwendeten Datentypen innerhalb der Implemeniterungsdateien *.cpp, also in der gelinkten Bibliothek stattfinden und das Interface samt den CASI-Header-Dateien nicht berühren.

[15] Der Zugriff auf die CASI-Modell-Parameter ist im Beispiel uneingeschränkt bezüglich aller GreenConfig-Parameter-Merkmale und in der zeitlichen Abfolge genau den GreenConfig-Parametern entsprechend.

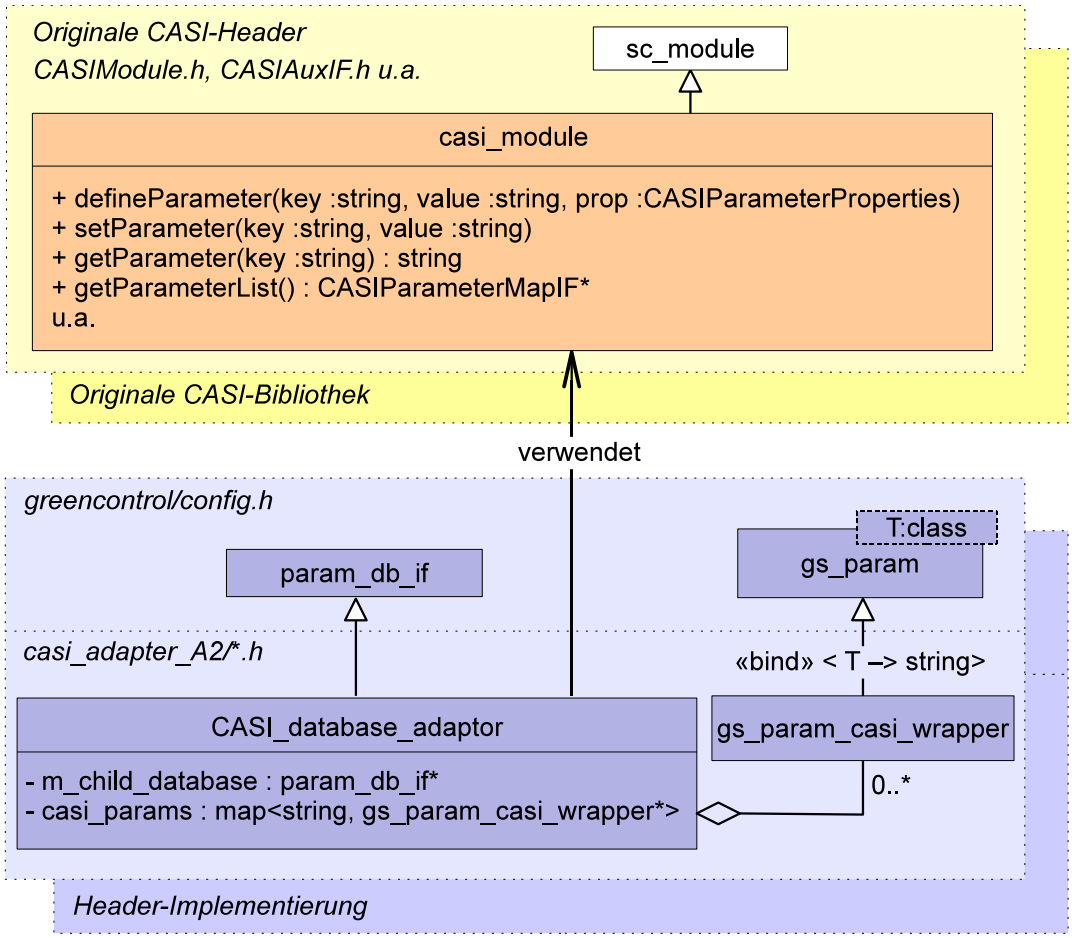


Abbildung 6.5.: PIU-CASI-Adapter für CASI-Parameter in einer GreenConfig-Umgebung (nach Adapter-Variante (f))

tionsmechanismus bereitstellt[16]. Dadurch hat der GreenConfig-Mechanismus keinen direkten Zugriff auf die Modell-Parameter. Folglich ist ein Adapter nach Variante (f) notwendig, der das zur Verfügung stehende proprietäre Konfigurations-Interface verwendet, um GreenConfig-Schattenparameter zu erstellen und zu pflegen.

Der Adapter ist zunächst unabhängig vom zu konfigurierenden Modul. Die Integration in GreenConfig erfolgt anhand einer angepassten Datenbank im Konfigurations-Plug-in (vgl. Abbildung 6.5). Der Datenbank-Adapter (Klasse `CASI_database_adaptor`) filtert sämtliche Zugriffe auf die dahinter liegende herkömmliche GreenConfig-Datenbank (Klassenmember `m_child_database`) und prüft, ob die Modell-Parameter, auf die vom Plug-in zugegriffen wird, vielleicht von einem CASI-Modul bereitgestellt werden. Ist das der Fall, wird beim ersten Zugriff ein Schattenparameter (vom Typ `gs_param_casi_wrapper`) angelegt, der unter Verwendung des CASI-Konfigurations-Interfaces seinen gespiegelten Wert bei jedem (Lese-

[16]Dieses Beispiel vereinfacht den Implementierungsaufwand für das Beispiel-CASI-Modul, indem es statt dem tatsächlichen Überladen der Konfigurations-Funktionen die unveränderte originale Implementierung verwendet. Aus Sicht des GreenConfig-Mechanismus hat die Verwendung der originalen unveränderten CASI-Bibliothek den gleichen Effekt, auch wenn das Modul das Interface tatsächlich nicht selbst implementiert. Deswegen wird das Beispiel mit der originalen CASI-Bibliothek gelinkt, wodurch der GreenConfig-Mechanismus (im Gegensatz zu der zuvor eingeführten Adapter-Variante) keinen Zugriff auf die Modell-Parameter hat.

und Schreib-)Zugriff automatisch mit dem Original synchronisiert. Ist ein solcher Schattenparameter angelegt, kann er ab sofort von jedem Tool oder Modell über die GreenConfig-API oder als GreenConfig-Parameter verwendet werden. Manipulative Wertzugriffe werden sofort an das CASI-Modul weitergegeben, wertneutrale Zugriffe werden vom Schattenparameter ausgeführt, nachdem der aktuelle Wert beim CASI-Modul abgefragt wurde.

Dieser Adapter ermöglicht folglich eine vollständige Synchronisation, da sämtliche Zugriffe auf den Modell-Parameter sowohl im originalen CASI-Modul als auch über den GreenConfig-Schattenparameter möglich und konsistent sind. Eine Einschränkung ist zu machen: Die Schattenparameter können keine garantierten Benachrichtigungen über Zugriffe auslösen, die innerhalb des CASI-Moduls oder dessen Konfiguṙations-Interface vorgenommen werden. Benachrichtigungen über manipulative Wertzugriffe erfolgen potentiell verspätet beim nächsten Zugriff auf den Schattenparameter, der die Wertänderung dann feststellt und benachrichtigen kann. Potentiell können diese Benachrichtigungen aber auch gar nicht stattfinden, wenn beispielsweise mehrere Änderungen innerhalb des CASI-Moduls erfolgen, bevor der Schattenparameter den Status prüft. Wertneutrale Zugriffe auf Seiten des CASI-Moduls können gar nicht auf GreenConfig-Seite benachrichtigt werden, da die Schattenparameter keine Möglichkeit haben, diese Zugriffe zu registrieren. Die (z.B. zusätzliche) Verwendung der oben beschriebenen Adapter-Variante (e) kann diesen Mangel beheben.

Um zu Demonstrationszwecken den Unterschied zur optimalen Umsetzung mit einem Adapter nach Variante (e) zu zeigen, kann das Beispiel auch mit einer wie oben beschrieben modifizierten CASI-Bibliothek gelinkt werden. Mit dieser Modifikation des Beispiels wird auch belegt, dass der globale Monitor bereits gespiegelte Parameter nicht doppelt anlegt.

UIP-Integration: Universelles GreenConfig-Modell in CASI-Umgebung

Die Realisierung der UIP-Integration demonstriert das Beispiel B.11, bei dem es sich um eine CASI-Simulation handelt[17]. Originale CASI-Module und GreenConfig-Module werden in einer Testbench gemischt. Das Beispiel zeigt, dass GreenConfig-Module über das CASI-Konfiguṙations-Interface konfigurierbar sind.

Das Beispiel verwendet das originale CASI 1.1.0 und demonstriert, dass ein GreenConfig-Modul mit kleinen Modifikationen in den proprietären Mechanismus eingebunden werden kann. Zu diesem Zweck muss jedes Modul, auch die GreenConfig-Module, die Klasse `casi_module` implementieren, in der in proprietären CASI-Modulen die Modell-Parameter verwaltet werden.

Ein GreenConfig-Modul besitzt dagegen Parameterobjekte, die nicht ins `casi_module` verschoben werden können. Allerdings ist es auf einfache Weise möglich, die Funktionsaufrufe des CASI-Konfigurations-Interfaces mit Hilfe der GreenConfig-API auf die GreenConfig-Parameter umzulenken. Diese Umsetzung realisiert der Adapter `gcnf_casi_adapter_mod` (vgl. Abbildung 6.6), der das als Adapter-Variante (h) in Abschnitt 3.3.3 vorgestellte Verfahren

[17] Die CASI-Simulation wird mangels CASI-Entwicklungsumgebung in einer OSCI-Umgebung durchgeführt.

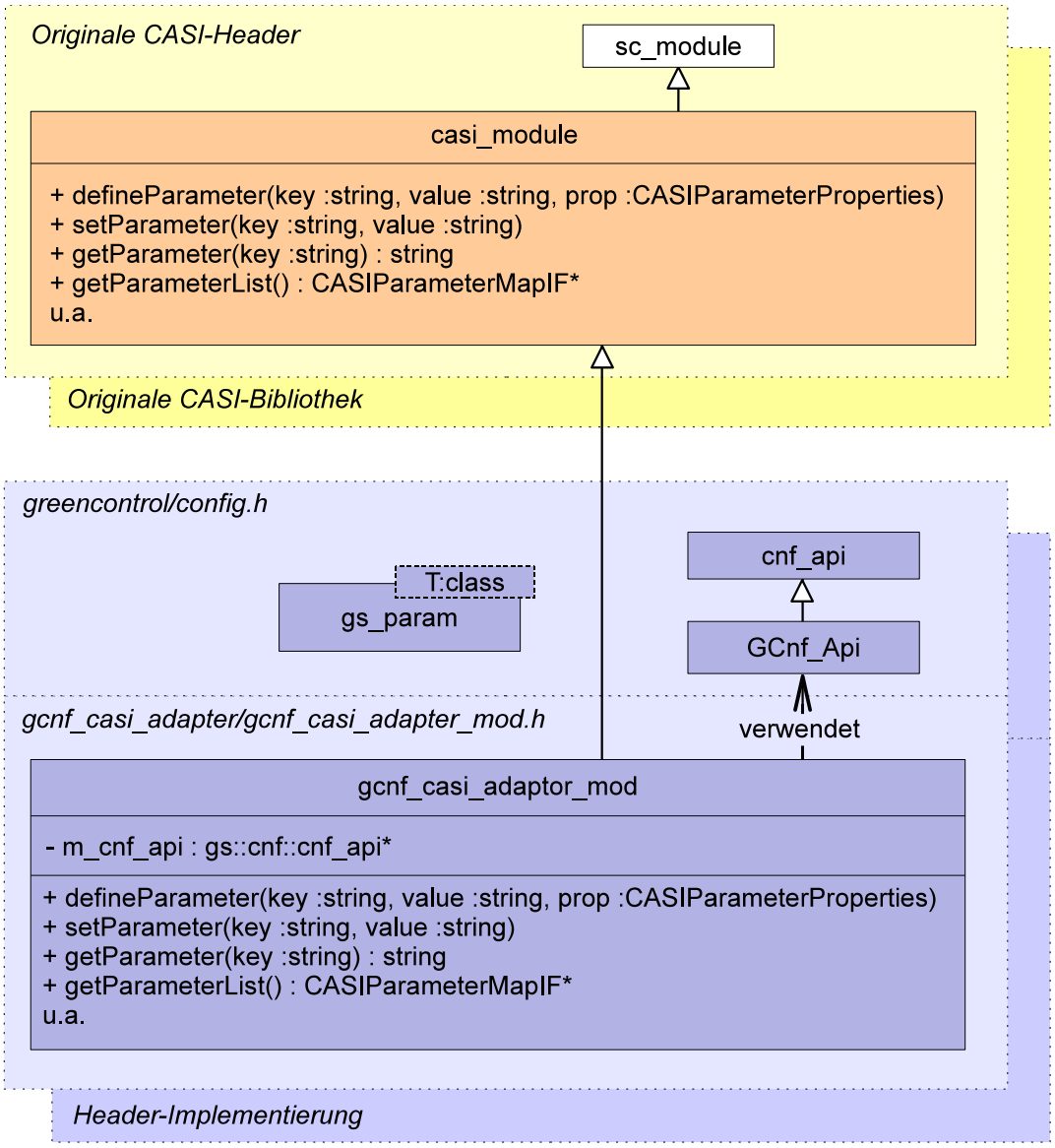


Abbildung 6.6.: UIP-CASI-Adapter für GreenConfig-Modelle in einer CASI-Umgebung (nach Adapter-Variante (h))

nutzt. In den Funktionen des Adapters werden die Zugriffe auf die moduleigenen GreenConfig-Parameter umgeleitet.

Der Adapter greift auf die moduleigenen Parameter über die GreenConfig-API zu (Member `m_cnf_api`). Deren Funktionen bekommen nur den lokalen Parameternamen übergeben, der mit dem Modulnamen als Präfix vervollständigt wird[18].

Folglich ist eine minimale Code-Modifikation der GreenConfig-Module notwendig: Sie leiten anstatt von `sc_module` vom Adapter `gcnf_casi_adapter_mod` ab[19].

[18]Damit funktioniert dieser Adapter nur für Parameter, die nicht manuell einen Top-Level-Namen zugewiesen bekommen haben.
[19]Der Adapter leitet wiederum von `sc_module` ab.

Fazit

Dieser Abschnitt hat die erfolgreiche Integration eines proprietären – den Konfigurations-Interface-Ansatz verfolgenden – Konfigurations-Mechanismus in den universellen Mechanismus demonstriert. Die erfolgreiche PIU-Integration mit einem Konfigurations-Interface-Mechanismus zeigt die Flexibilität GreenConfigs und des dort verwendeten Class-Wrapper-Ansatzes. Der Erfolg wird verstärkt durch die gelungene UIP-Integration, die nur weniger unumgänglicher Modifikationen am Modell bedarf.

6.4. Nachbildung hierarchischer Rangfolge

Dieser Abschnitt stellt ein Verfahren vor, mit dem Modelle die hierarchische Konfigurationsrangfolge verwenden können, dabei aber mit dem universellen Konfigurationsmechanismus GreenConfig konfiguriert werden, der die zeitliche Konfigurationsrangfolge realisiert (vgl. Abschnitt 3.1.3). Die hierarchische Rangfolge wird folglich mit der zeitlichen nachgebildet.

Das hier diskutierte Vorgehen ist im Beispiel B.2 zu finden. Es wendet die Rangfolgenintegration 2 nach Abschnitt 3.3.4 an, das heißt es stellt die hierarchische Konfigurationsrangfolge für den manipulativen Wertzugriff vom Typ „Setzen von Initialwerten“, den wichtigsten Anwendungsfall, sicher: Die Initialwerte werden von verschiedenen Modulen mit der Funktion `setInitValue` der GreenConfig-API gesetzt und mit der Funktion `lockInitValue` gesperrt. Erst im Anschluss wird das Subsystem erzeugt.

In Abbildung 6.9 auf Seite 144 ist der Aufbau des Beispiels als Objektdiagramm illustriert. Die gebogenen (roten) Pfeile deuten an, welche Modulinstanz welchen Initialwert eines Parameters setzt. Die mit gestrichelten Linien dargestellten Initialwerte werden von einer Konfiguration höherer Rangfolge überschrieben.

Alle Parameter sind vom Typ String, damit sie zu Demonstrationszwecken jeweils im eigenen Wert dessen Herkunft speichern können. Die Werte werden von den konfigurierenden Modulen auf deren Modulnamen und den Typ des manipulativen Wertzugriffs, mit dem der Wert geschrieben wurde, gesetzt. Als Nachweis der korrekten Funktionsweise erzeugt die Testbench am Ende der Simulation eine Ausgabe, die die Modell-Parameter mit den tatsächlich zugewiesenen Werten auflistet (siehe Auszug in Listing 6.10 auf Seite 144).

Zur Erläuterung soll exemplarisch die Konfiguration eines Modell-Parameters herausgegriffen werden: Listing 6.7 zeigt den Konstruktor des hierarchisch höchsten Moduls ModuleATop1 (Klasse `ModuleATop`). Im Konstruktor wird der Initialwert des Parameters „ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD2.my_param3“ auf „Init value set by ModuleATop1“ gesetzt (Zeile 4) und anschließend gesperrt (Zeile 5). Nachfolgend wird das Subsystem erzeugt (Zeile 6). Dieser Parameterwert bleibt bestehen (vgl. Listing 6.10 Zeile 6), obwohl das hierarchisch niedrigere Modul ModuleATop1.ModuleBSubsystem1.ModuleC1 den Initialwert des gleichen Parameters zu einem späteren Zeitpunkt zu überschreiben versucht (gestrichelte Linie in Abbildung 6.9). Listing 6.8 zeigt einen Teil des Modul-Konstruktors. Der wegen der bereits bestehenden Sperre erfolglose Versuch, den Parameterwert zu setzen, wird in den

```
1 ModuleATop(sc_module_name name) : sc_module(name) {
2   m_cnf_api = gs::cnf::GCnf_Api::getApiInstance(this); [...]
3   string val = "Init value set by "; val += sc_module::name();
4   m_cnf_api->setInitValue ↩
        ("ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD2.my_param3", val);
5   m_cnf_api->lockInitValue ↩
        ("ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD2.my_param3");
6   m_subsystem = new ModuleBSubsystem("ModuleBSubsystem1");
7 }
```

Listing 6.7: Konstruktor vom hierarchisch höchsten Modul (gekürzt)

```
1 ModuleC(sc_module_name name) : sc_module(name) { [...]
2   std::string val = "Init value set by "; val += sc_module::name();
3   [...]
4   m_cnf_api->setInitValue ↩
        ("ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD2.my_param3", val);
5   m_cnf_api->lockInitValue ↩
        ("ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD2.my_param3");
6   [...]
7   m_submoduleD2 = new ModuleD("ModuleD2");
8 }
```

Listing 6.8: Konstruktor vom hierarchisch niedrigeren Modul (gekürzt)

Zeilen 4 und 5 vorgenommen. Auch hier wird das Submodul „ModuleD2“ erst im Anschluss erzeugt.

Fazit

In diesem Abschnitt wird ein Beispiel beschrieben, das mit dem universellen Konfigurationsmechanismus GreenConfig die hierarchische Konfigurationsrangfolge nachbildet. Damit wird gezeigt, dass der universelle Konfigurationsmechanismus im Bereich der Rangfolgen für das Hauptziel dieser Arbeit geeignet ist, die Austauschbarkeit von Modellen zu verbessern. Die hier beschriebene Herangehensweise kann genutzt werden, um Adapter für die PIU-Integration zu entwickeln.

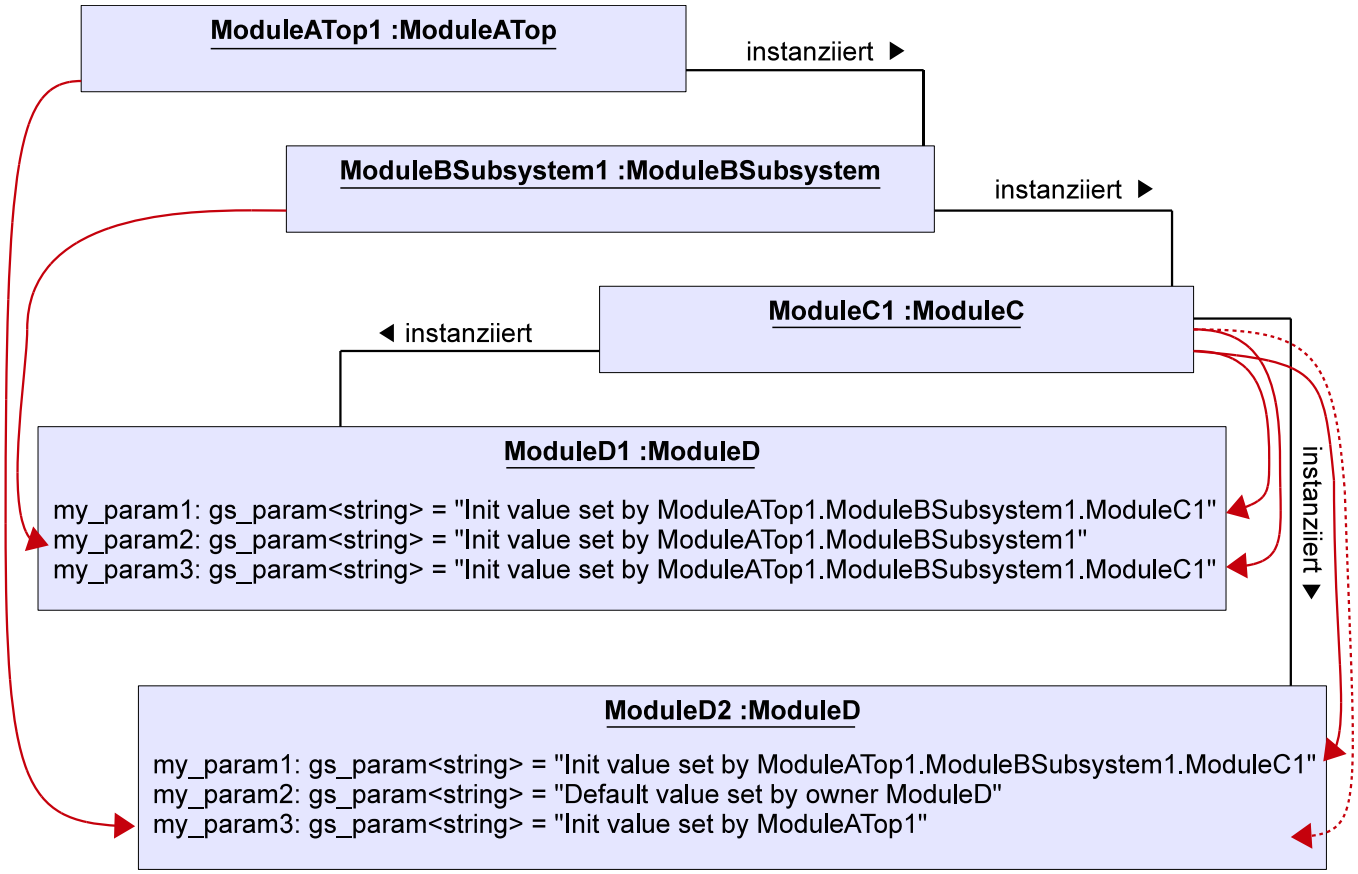


Abbildung 6.9.: Beispiel hierarchische Rangfolge (Objektdiagramm)

```
1 string : ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD1.my_param1 = Init value ⤸
      set by ModuleATop1.ModuleBSubsystem1.ModuleC1   , no attributes
2 string : ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD1.my_param2 = Init value ⤸
      set by ModuleATop1.ModuleBSubsystem1            , no attributes
3 string : ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD1.my_param3 = Init value ⤸
      set by ModuleATop1.ModuleBSubsystem1.ModuleC1   , no attributes
4 string : ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD2.my_param1 = Init value ⤸
      set by ModuleATop1.ModuleBSubsystem1.ModuleC1   , no attributes
5 string : ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD2.my_param2 = Default ⤸
      value set by owner ModuleD                      , no attributes
6 string : ModuleATop1.ModuleBSubsystem1.ModuleC1.ModuleD2.my_param3 = Init value ⤸
      set by ModuleATop1                              , no attributes
```

Listing 6.10: Ausgabe Beispiel hierarchische Rangfolge (Auszug)

6.5. Adapter für die Open-Verification-Methodology

Dieser Abschnitt beschreibt die Integration des Konfigurationsteils der in Abschnitt 2.6.6 eingeführten Open-Verification-Methodology (OVM) in den universellen Konfigurationsmechanismus GreenConfig. Besondere Aufmerksamkeit legt dieser Abschnitt auf die Integration der hierarchischen Rangfolge in die zeitliche Rangfolge des universellen Mechanismus (vgl. Abschnitte 3.1.3 und 3.3.4). OVM verwendet Konfigurations-Interfaces, die im Rahmen dieser Integration auf den Class-Wrapper-Ansatz umgesetzt werden, was hier allerdings nicht detailliert erläutert wird (siehe dazu beispielsweise Abschnitt 6.3).

Das Beispiel B.12 verwendet modifizierten OVM-Code, der – wie das Original auch – mit dem dort enthaltenen Anwendungsbeispiel OVM_example_fuer_mod[20] getestet werden kann. Der Adapter (ovm-2.0.1_ml_mod_2) wendet im Gegensatz zum vorangehenden Abschnitt 6.4 die Rangfolgenintegration 3 aus Abschnitt 3.3.4 an: Da OVM frei verfügbar ist, kann der bestehende hierarchische Mechanismus weiter genutzt werden und intern mit einer Verbindung zu GreenConfig ausgestattet werden. Dadurch wird erreicht, dass sämtliche in OVM unterstützten Merkmale, wie beispielsweise die semantisch identische Interpretation von Wildcards sowie die hierarchische Rangfolge selbst, original erhalten bleiben. Lediglich die Speicherung der Modell-Parameter wird an die des GreenConfig-Mechanismus angebunden. Es wird davon ausgegangen, dass sich das Modell an die Empfehlung der OVM-Dokumentation hält, die Konfiguration von Parametern eines Subsystems vor dessen Erzeugung durchzuführen.

Zunächst wird die existierende OVM-Implementierung übernommen und modifziert (im Wesentlichen betrifft das die Dateien ovm_config.*). Da das in OVM empfohlene Verfahren vorausgesetzt wird, jeden Schreibvorgang auf Parameterwerte vor dem ersten Lesen dieser Werte durchzuführen, entspricht das Setzen eines OVM-Parameters dem Setzen eines Initialwertes in GreenConfig. Erst das Lesen des Wertes lässt den OVM-Parameter einen expliziten Parameter werden.

Folglich wird im modifizierten OVM beim Setzen von Parameterwerten in den Funktionen `set_config_[int|string]`[21] der Wert zunächst nur OVM-intern gespeichert und beim Abruf des Wertes in der gewohnten hierarchischen Konfigurationsrangfolge wieder ausgelesen.

Sobald ein OVM-Parameter das erste Mal gelesen wird, wird der entsprechende GreenConfig-Parameter in den Funktionen `get_config_[int|string]_internal` explizit erzeugt. Dabei wird dem GreenConfig-Parameter-Konstruktor der innerhalb des OVM-Mechanismus gespeicherte Initialwert als Defaultwert übergeben. Ein in GreenConfig gesetzter Initialwert wird diesen Wert überschreiben. Dieses Verhalten ist beabsichtigt, da GreenConfig bei dieser Integration von OVM in eine GreenConfig-Umgebung Vorrang hat.

Die OVM-interne Managerklasse `ovm_config_mgr` wird erweitert um private Variablen für die Speicherung der GreenConfig-API und der erzeugten GreenConfig-Parameter, damit letztere zum Ende der Simulation wieder vorschriftsmäßig im Destruktor gelöscht werden können. Zusätzlich bekommt die Klasse eine Callback-Funktion, die bei Änderungen an den

[20] Das Beispiel basiert auf einem von OVM mitgelieferten Beispiel.
[21] Diese Proof-of-Concept-Implementierung behandelt nicht den Objekt-Typ.

GreenConfig-Parametern aufgerufen wird, um diese an die OVM-Parameter weiterzugeben bzw. Fehler zu generieren[22].

Der Adapter prüft auf einige unerwartete Zustände. Beispielsweise wird beim Setzen von OVM-Parameterwerten kontrolliert, ob der Wert noch nicht ausgelesen wurde und folglich als expliziter GreenConfig-Parameter existiert. Das würde den OVM-Empfehlungen widerstreben. Zusätzlich wird überprüft, ob beim Abruf eines Wertes über die OVM-API bereits ein entsprechender GreenConfig-Parameter existiert. Das trifft zu, wenn der OVM-Wert mehrfach abgerufen wird – was erlaubt ist. Allerdings wird zusätzlich geprüft, ob die Werte in OVM und dem GreenConfig-Parameter identisch sind. Wenn beim Erzeugen des GreenConfig-Parameters ein GreenConfig-Initialwert den OVM-Initialwert, übergeben als Konstruktor-Defaultwert, überschreibt, wird der neue Wert anschließend von OVM zurückgegeben. Für den einmal erzeugten GreenConfig-Parameters wird sichergestellt, dass dessen Wert nach dem ersten Abruf des OVM-Wertes nicht mehr geändert wird.

Fazit

Der hier beschriebene Adapter ermöglicht die PIU-Integration eines OVM-Subsystems in eine GreenConfig-Simulation, solange das OVM-Modell die Empfehlungen der OVM-Dokumentation befolgt. OVM-Parameter können mit GreenConfig-Initialwerten gesetzt und als GreenConfig-Parameter ausgelesen werden. Sie behalten innerhalb des OVM-Systems ihre hierarchische Präzedenz bei.

6.6. Großes Integrationsbeispiel

In den vorangehenden Abschnitten wurden verschiedene Integrationen kommerzieller Konfigurationsmechanismen vorgestellt. Dieser Abschnitt beschreibt ein System, in dem alle Konfigurationsmechanismen zusammenarbeiten.

Das System besteht aus SystemC-Modulen, die im Zusammenhang mit den Entwicklungsumgebungen der verschiedenen Hersteller entwickelt wurden. Sie verwenden folglich jeweils unterschiedliche Konfigurationsmechanismen: GreenConfig-Parameter, CCSS-Parameter, SCML-Properties und OVM-Parameter. Das Ziel ist, das System mit einem einzigen Mechanismus zu konfigurieren und gemeinsam zu simulieren. Damit wird die Austauschbarkeit der Modelle als Hauptziel dieser Arbeit demonstriert.

Die Integration wird in zwei Teilen gezeigt: Zum einen wird das System in einer OSCI-Umgebung simuliert und mit dem universellen Mechanismus GreenConfig konfiguriert. Zum anderen wird das gleiche System unverändert im CoWare Platform-Creator simuliert und konfiguriert. Das verwendete System sowie die Projektdateien für beide Teile sind im Beispiel B.13 zu finden.

[22] Eine Parameteränderung nachdem ein Parameter explizit erzeugt wurde, also innerhalb OVMs mit get ausgelesen wird, soll laut OVM-Empfehlung nicht vorkommen.

Das System besteht aus den Modulen GCnf_Mod (Klasse `Gs_Orig_IP`), SCML_Mod (Klasse `Scml_Orig_IP`), SCML_WIZ_Mod (Klasse `wiz_scml_example`), OVM_Mod (Klasse `top`) und CCSS_Mod (Klasse `Top_CCSS_Module`). An den Bezeichnern ist jeweils abzulesen, welchen Konfigurationsmechanimus das Modul verwendet. Das Modul SCML_Mod wurde manuell für den SCML-Mechanismus entwickelt, wogegen SCML_WIZ_Mod ein vom CoWare-Wizard automatisch generiertes Modul ist. Die Module CCSS_Mod und OVM_Mod bestehen aus hierarchischen Subsystemen, deren Submodule ebenfalls konfigurierbar sind. Das zusätzliche Modul Observer (Klasse `Gcnf_Observer`) erlaubt das Nachvollziehen sämtlicher Parameterzugriffe, solange die entsprechende Benachrichtigung vom Adapter zur Verfügung gestellt wird. Der Observer kann mit einem GreenConfig-Parameter aktiviert oder deaktiviert werden. Die Module selbst sind konfigurierbar, erfüllen aber keine Funktion.

OSCI-Simulation mit GreenConfig-Konfiguration

Der erste Teil des Beispiels simuliert das System mit dem OSCI-Kernel und dem universellen Konfigurationsmechanismus GreenConfig. Die verschiedenen Module verwenden die jeweiligen Adapter, sodass alle Modell-Parameter transparent mit GreenConfig konfigurierbar sind. In der Testbench werden einige der Parameter[23] mit Initialwerten konfiguriert, die im weiteren Verlauf der Simulation nicht überschrieben werden, also in der Ausgabe zu beliebigen Zeitpunkten sichtbar werden.

Der Screenshot in Abbildung 6.11 auf der nächsten Seite zeigt als Ergebnis alle in der Simulation vorhandenen GreenConfig-Parameter und ihre Werte[24]. Alle Parameter werden unabhängig von ihrer Art und mit dem richtigen Wert angezeigt.

Alle Merkmale, die der jeweils originale Mechanismus bereitstellt und die vom Adapter umgesetzt werden, sind auch unter GreenConfig verfügbar. Beispielsweise können die Parameter im abgebildeten Fenster gesetzt werden, soweit der entsprechende Adapter eine Wertänderung während der Simulationsphase unterstützt[25].

Die hier demonstrierte PIU-Integration erfolgt für alle Merkmale der proprietären Mechanismen, womit die Anforderung AA7 für alle Mechanismen und Adapter erfüllt ist.

CoWare Platform-Creator

Der zweite Teil des Beispiels simuliert das identische System im Platform-Creator mit dem dort enthaltenen proprietären SystemC-Kernel und dem Konfigurationsmechanimus SCML bzw. der dafür vorgesehenen grafischen Oberfläche.

Abbildung 6.12 auf Seite 150 zeigt das System im Platform-Creator. Die Module sind im Register „Design“ dargestellt. Die Modell-Parameter des jeweils ausgewählten Moduls werden im Parameter-Editor mit ihrem Initialwert angezeigt. Abbildung 6.13 auf Seite 151

[23] Die mit Initialwerten konfigurierten Parameter sind GCnf_Mod.m_stay_init_par, SCML_Mod.m_stay_init_prop, CCSS_Mod.stay_init_par, OVM_Mod.inst1.u2.hierarchtest.
[24] Der Screenshot zeigt das Tool SCRSI, das in Abschnitt 6.7 vorgestellten wurde.
[25] Der OVM-Adapter gibt in diesem Fall eine Warnung aus, die anderen Adapter erlauben das.

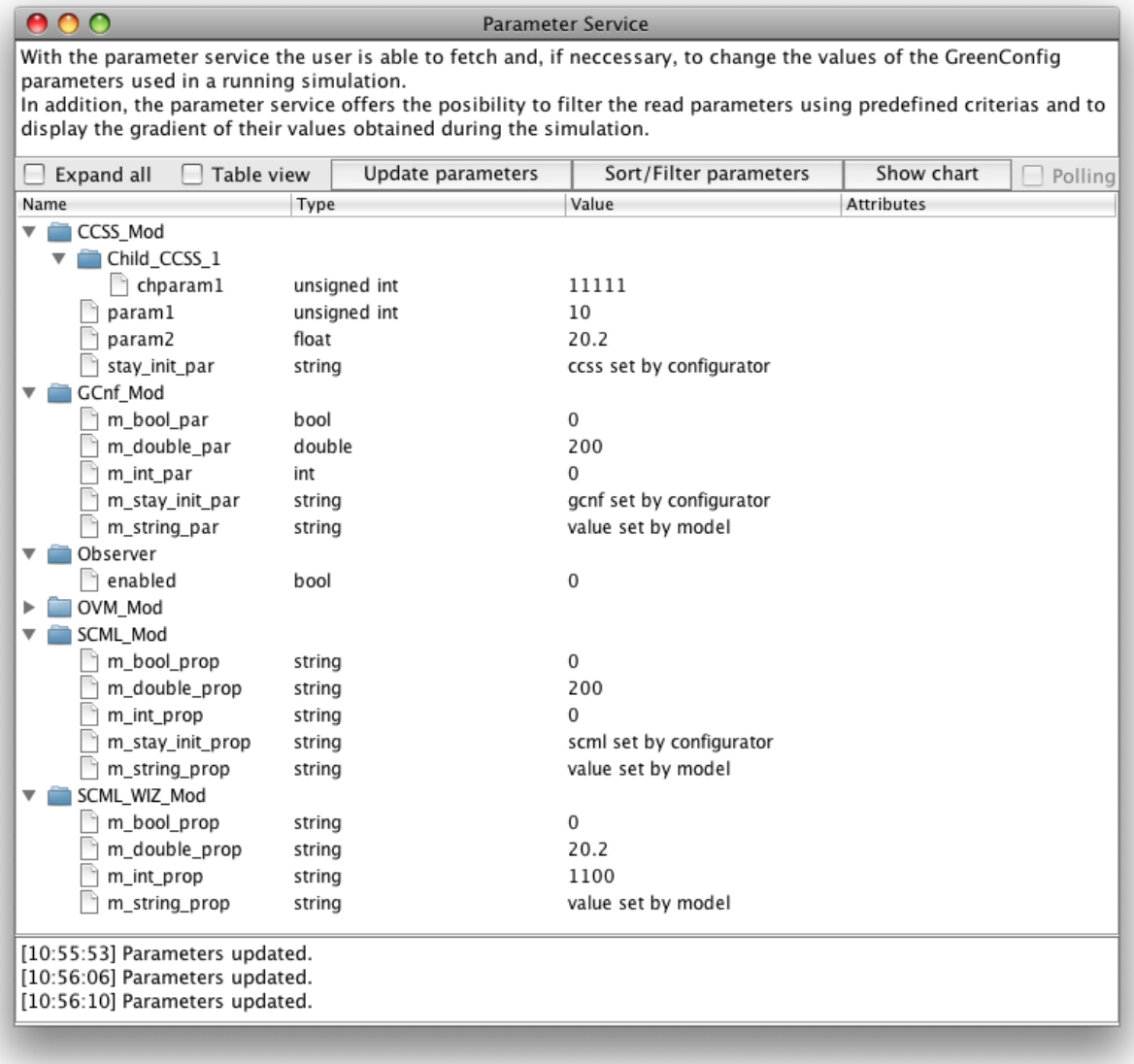


Abbildung 6.11.: Konfiguration des Beispielsystems in einer OSCI-Simulation mit SCRSI (Screenshot)

zeigt alle Parameter-Editoren für die verschiedenen Module. Hier werden die Initialwerte aller Parameter identisch mit der OSCI-Simulation (erster Teil) gesetzt. Das zeigt zusammen mit der (nicht dargestellten) Kontrolle der Simulationsausgabe, die der Ausgabe der OSCI-Simulation gleicht, dass alle Parameter korrekt integriert sind und über die Oberfläche gesetzt werden können.

Die hier demonstrierte UIP-Integration ist durch die Merkmale der Platform-Creator-IDE eingeschränkt:

Beispielsweise können während der Simulationphase keine Zugriffe auf Modell-Parameter über die grafische Oberfläche erfolgen. Je nach Art des Adapters arbeitet intern der GreenConfig-Mechanismus und kann in dem Fall zusätzliche Merkmale ergänzen. Ein Beispiel dafür sind SCML-Parameter, deren Wertänderung während der Simulationsphase über GreenConfig möglich ist, nicht aber über die Oberfläche des Platform-Creators.

Eine weitere Einschränkung ist das Konfigurieren von Parametern, die hierarchisch tiefer liegen als das im Platform-Creator angezeigte Modul (siehe Abbildung 6.13(d)). Ein solcher Parameter muss der Anzeige über einen Skript-Befehl hinzugefügt werden, anstatt ihn im GUI erzeugen zu können.

Fazit

Dieses Integrationsbeispiel demonstriert das Erreichen des Hauptziels dieser Arbeit, die Austauschbarkeit von Modellen zwischen verschiedenen Entwicklungsumgebungen bezüglich der Konfiguration zu ermöglichen. Der erste Teil zeigt die vollständig transparente Integration der Modelle in eine OSCI-Umgebung mit GreenConfig-Konfiguration, also die PIU-Integration. Der zweite Teil zeigt, dass sogar die Integration und Konfiguration des Systems in die proprietäre IDE Platform-Creator erfolgreich gelungen ist – unter gewissen Einschränkungen, die durch die Fähigkeiten des proprietären Mechanismus hervorgerufen werden. Das heißt für die exemplarisch verwendete IDE ist auch die UIP-Integration erfolgreich gewesen. Sogar die Konfiguration proprietärer Modelle in einer proprietären Simulation eines anderen Herstellers (PIU- + UIP-Integration, vgl. Abbildung 3.10 auf Seite 50) ist über den Umweg des universellen Mechanismus erfolgreich.

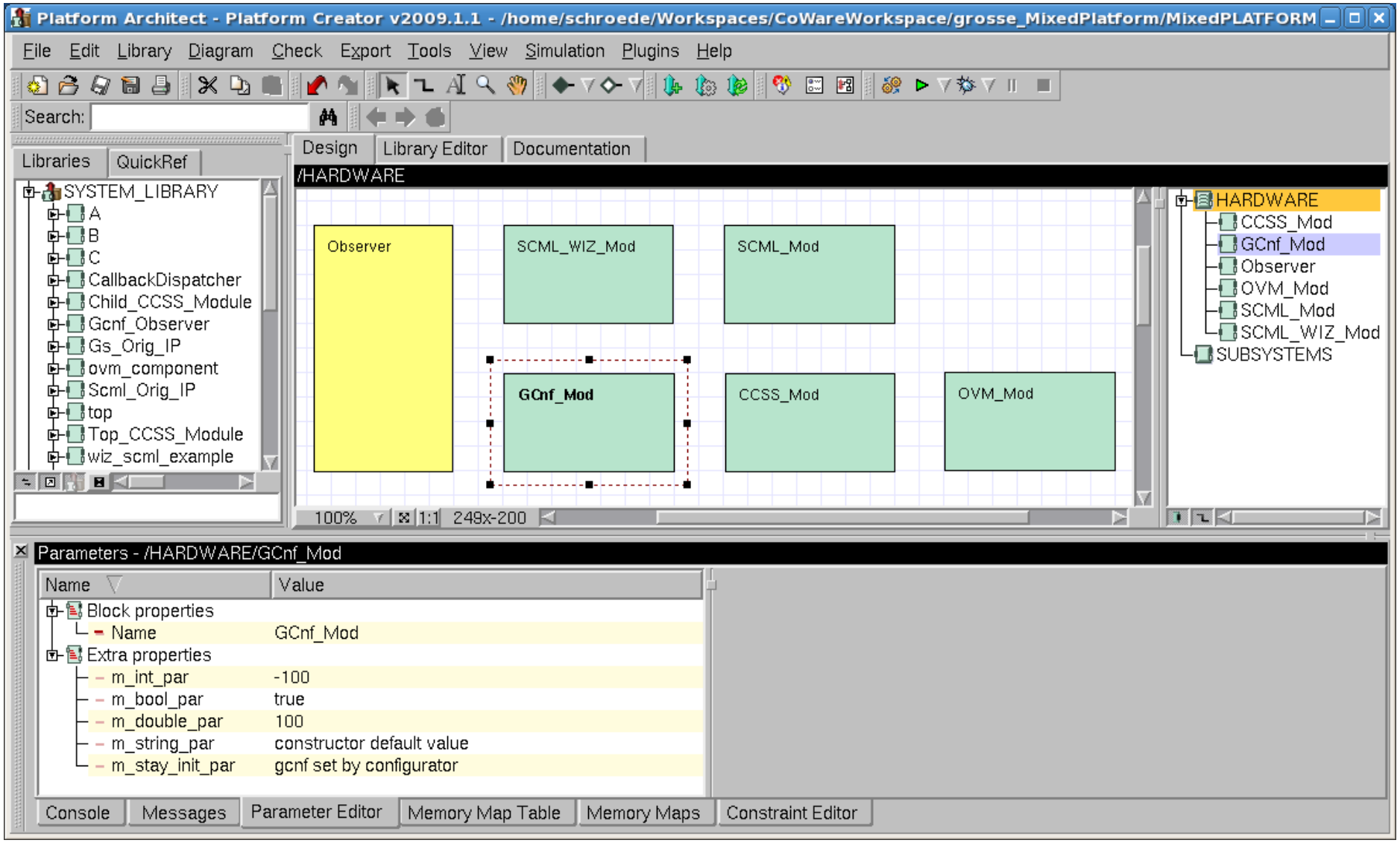


Abbildung 6.12.: Projekt im Platform-Creator; Integration verschiedener Modelle (Screenshot)

Parameters - /HARDWARE/Observer

Name	Value
Block properties	
Name	Observer
Extra properties	
enabled	false

(a) Modul Observer

Parameters - /HARDWARE/GCnf_Mod

Name	Value
Block properties	
Name	GCnf_Mod
Extra properties	
m_int_par	-100
m_bool_par	true
m_double_par	100
m_string_par	constructor default value
m_stay_init_par	gcnf set by configurator

(b) Modul GCnf_Mod

Parameters - /HARDWARE/SCML_Mod

Name	Value
Block properties	
Name	SCML_Mod
Extra properties	
m_int_prop	-100
m_bool_prop	false
m_double_prop	100
m_string_prop	constructor default value
m_stay_init_prop	scml set by configurator

(c) Modul SCML_Mod

Parameters - /HARDWARE/OVM_Mod

Name	Value
Block properties	
Name	OVM_Mod
Extra properties	
inst1.u2.hierarchtest	ovm set by configurator

(d) Modul OVM_Mod

Parameters - /HARDWARE/SCML_WIZ_Mod

Name	Value
Block properties	
Name	SCML_WIZ_Mod
Constructor Arguments	
m_int_prop	1000
m_bool_prop	true
m_string_prop	Wizard default value
m_double_prop	10.09999999999999964

(e) Modul SCML_WIZ_Mod

Parameters - /HARDWARE/CCSS_Mod

Name	Value
Block properties	
Name	CCSS_Mod
Extra properties	
param1	10
param2	20.200000e+00
stay_init_par	ccss set by configurator

(f) Modul CCSS_Mod

Abbildung 6.13.: Modell-Parameter als SCML-Properties im Platform-Creator (Screenshots)

6.7. SystemC-Remote-Service-Interface (SCRSI)

Das SystemC-Remote-Service-Interface (SCRSI) ist ein Tool, das mit einer in Java entwickelten grafischen Oberfläche Zugriff auf die SystemC-Simulation erlaubt. Das Tool ist modular um Services für Meta-Funktionalität erweiterbar. SCRSI ist ein studentisches Projekt, das in verschiedenen Arbeiten entstanden ist [Meie08, Tutt09, Jesk10, Schm11]. Der Code befindet sich im externen Listing B.14. Es gibt verschiedene Services, überwiegend ermöglichen sie den Zugriff auf Funktionen und Plug-ins des GreenControl-Projekts:

- *Simulationskontrolle* Ein von GreenControl unabhängiger Service ist die Simulationskontrolle. Sie gibt dem Benutzer Kontrolle über den zeitlichen Ablauf der Simulation. Sie kann gestoppt, pausiert und fortgesetzt werden oder bis zu einem ausgewählten Simulationszeitpunkt weiterlaufen. Außerdem können Pausen aus der Simulation heraus ausgelöst werden. Abbildung 6.14 zeigt einen Screenshot.

- *Parameterservice* Damit können alle GreenConfig-Parameter zu beliebigen Simulationszeitpunkten angezeigt und manipuliert werden. Der Werteverlauf von Parametern kann als Graph visualisiert werden. Abbildung 6.11 auf Seite 148 zeigt einen Screenshot.

- *Breakpointservice* Hier können Breakpoints mit verschiedenen Auslösebedingungen für beliebige Parameter definiert werden. Beim Eintreten einer Bedingung pausiert die Simulation mit Hilfe der Simulationskontrolle.

- *Berechnungsservice* Dieser erlaubt das Erstellen von sich stetig aktualisierenden Kalkulatoren, die Parameter miteinander verrechnen (vgl. Anhang A). Die Ergebnisse können grafisch visualisiert werden oder als Bedingung für Breakpoints dienen.

- *Konfigurationsdateiservice* Als Erweiterung des Parameterservice erlaubt der Konfigurationsdateiservice das Einlesen und Anwenden von GreenConfig-Konfigurationsdateien.

Aufbau

SCRSI basiert auf einer Client-Server-Architektur: Die SystemC-Simulation enthält einen zusätzlichen Server-Thread, der auf die Verbindung eines Clients wartet. Der Client ist die grafische Benutzeroberfläche und ist in Java realisiert. Die Kommunikation wird über eine TCP/IP-Verbindung durchgeführt. Abbildung 6.15 zeigt diesen grundlegenden Aufbau. Details können der Studienarbeit [Schm11] und den weiteren studentischen Arbeiten [Meie08, Tutt09, Jesk10] entnommen werden.

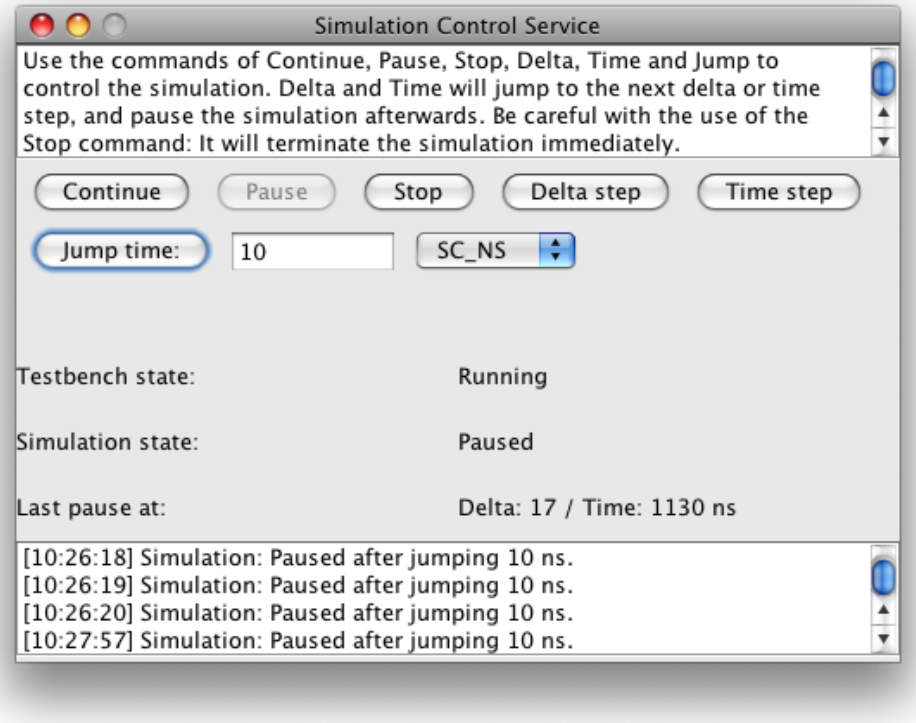


Abbildung 6.14.: SCRSI Simulationskontrolle (Screenshot)

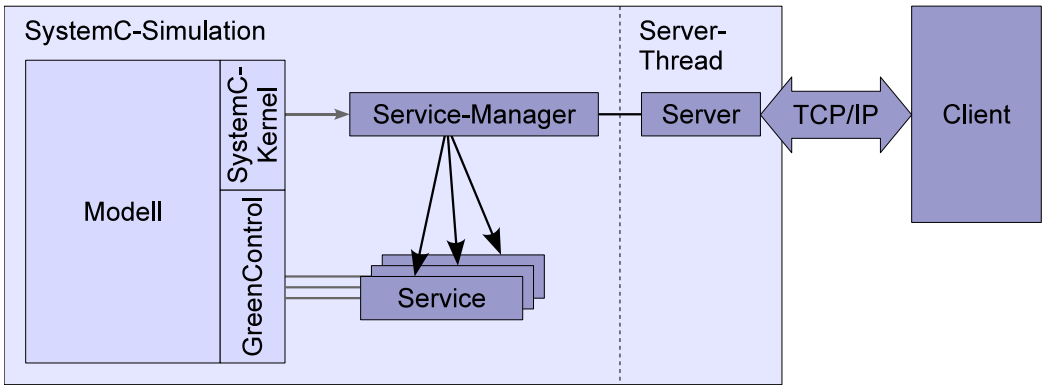


Abbildung 6.15.: SCRSI Client-Server-Architektur

Server

Der SystemC-seitige Teil von SCRSI kann in der Testbench in die Simulation eingebunden und initialisiert werden: Das Einbinden der Datei scrsi.h lädt die Definitionen des Hauptteils und die der existierenden Services. Anschließend können die benötigten Services je nach Bedarf initialisiert werden. Die Simulation muss mit der SCRSI-Bibliothek gelinkt werden.

Der Service-Manager verwaltet die verschiedenen Services und empfängt und versendet Nachrichten über den Server. Der Server läuft in einem eigenen System-Thread (nicht zu verwechseln mit SystemC-Threads), ist also echt nebenläufig zur Simulation und sichert somit die unterbrechungsfreie Kommunikation mit dem Client. Der SCRSI-Server unterbricht die Programmausführung während seiner Initialisierung und wartet auf die Verbindung eines Clients.

SCRSI soll möglichst unabhängig vom verwendeten SystemC-Kernel sein. Deswegen baut es wo immer möglich auf dem standardisierten SystemC [IEEE06] auf. Lediglich die Simulationskontrolle benötigt ein vom Standard nicht bereitgestelltes Merkmal, das allerdings portabel gelöst wurde: Unterbrechungen der Simulation werden im OSCI-Kernel mit Trace-File-Callbacks realisiert, in dem eine Schleife ausgeführt wird. Da jeder SystemC-Kernel eine solche oder ähnliche Funktion haben sollte, dürfte eine Anpassung in den meisten Fällen auch für proprietäre Kernels möglich sein[26].

Client

Der SCRSI-Client muss nach dem Start mit einer Simulation verbunden werden. Der TCP-Port kann beliebig angegeben werden, die Simulation kann also auf einem entfernten Rechner ausgeführt werden. Anschließend bietet der Client in seinem Hauptfenster eine Übersicht über die in der Simulation zur Verfügung stehenden Services, die sich jeweils in einem eigenen Fenster öffnen (vgl. Screenshot in Abbildung 6.16).

Fazit

SCRSI stellt eine grafische Oberfläche für die Konfiguration verschiedener in GreenConfig integrierter Konfigurationsmechanismen zur Verfügung. Damit leistet es einen wichtigen Beitrag zur Konfigurations-Interoperabilität. Ist es kompatibel zum proprietären Kernel einer IDE oder wird an diesen angepasst, kann es dort sogar für die Modell-Konfiguration und -Steuerung verwendet werden.

6.8. Weitere Anwendungsbeispiele

Dieser Abschnitt gibt einen kurzen Überblick über einige weitere Beispiele und Projekte, in denen Mechanismen der vorangehenden Kapitel verwendet werden.

[26] Wenn der proprietäre Kernel auf dem OSCI-Kernel aufbaut, ist die Wahrscheinlichkeit hoch, dass die Trace-File-Callbacks unverändert existieren.

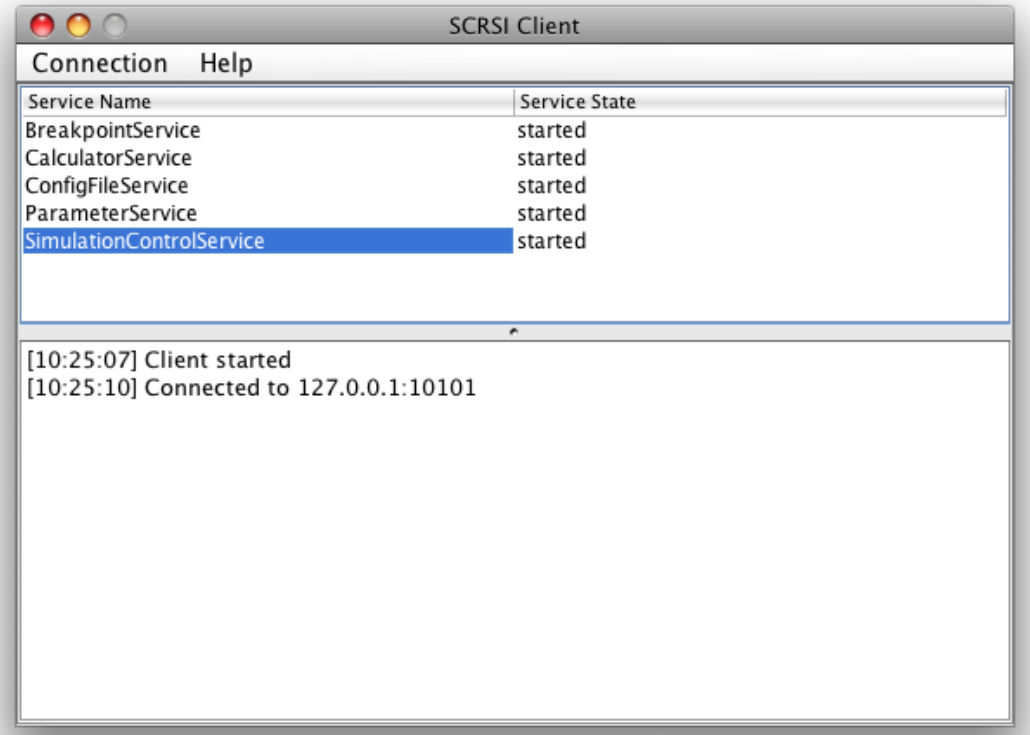


Abbildung 6.16.: SCRSI Client-Hauptfenster (Screenshot)

6.8.1. Konfigurations-Anwendungsstudie GreenReg

Das Projekt GreenSocs-Registers (GreenReg) ist eine Weiterentwicklung des Device Register Framework (DRF) von Intel. Eine wesentliche Funktion ist das Bereitstellen von mächtigen Registern für die Hardware-Modellierung. Diese bereits im Ursprungsprojekt DRF vorhandenen Register sind in GreenReg erweitert, sodass sie nun die GreenConfig-Parameter-Schnittstelle besitzen. Somit sind sie von allen Werkzeugen konfigurierbar, die mit GreenConfig oder einem Adapter kompatibel sind. Der Quellcode für dieses komplexe Konfigurationsbeispiel kann in [Schr10a][i] heruntergeladen werden. Details enthält [Schr10b][i].

Ein Beleg für den kommerziellen Einsatz dieses Projekts ist eine von CircuitSutra Technologies entwickelte Virtuelle Plattform der Open Core Protocol International Partnership (OCP-IP), die das „OCP Modeling Kit“ (OMK) verwendet [OCPI10][i]. Die Virtuelle Plattform kann kostenlos per Download bezogen werden[27]. Diese Plattform verwendet GreenReg-Register und ist folglich einerseits mit dem universellen Konfigurationsmechanismus GreenConfig und andererseits über Adapter mit proprietären Mechanismen konfigurierbar.

6.8.2. Checkpoint and Restore

In [MES+10] und [Mont10] werden GreenConfig-Parameter für das Checkpointing von SystemC-Simulationen verwendet. In dem Projekt wird der Zustand einer (eingeschränkten) Sys-

[27]Download der Virtuellen Plattform unter `http://www.ocpip.org/vp_package.php`.

Aspekt		**relevant für**	
		Konfiguration	Analyse
Existenz	Middleware mit Services	+	+
	einzelne Parameter	+	+
manipulative Wertzugriffe	von außen (Konfiguration)	◯	◯
	durch das Modell	◯	+
wertneutrale Zugriffe	von außen (Untersuchung)	◯	+
	durch das Modell	◯	+

Tabelle 6.17.: Performanceaspekte während der Simulationsphase
+: relevant, ◯ eingeschränkt relevant

temC-Simulation vollständig von GreenConfig-Parametern repräsentiert. Zu diesem Zweck werden ein modifizierter Kernel zur Verfügung gestellt und Einschränkungen für die Modelle vorgegeben: Modelle dürfen keine Threads (`SC_THREAD`s), sondern nur Methoden (`SC_METHOD`s) verwenden und müssen alle Zustände in Parametern speichern.

Die Fähigkeiten der GreenConfig-Parameter sorgen hier dafür, dass ein Snapshot der Simulation mit einem einfachen Export aller im System befindlichen Parameter vollständig beschrieben ist (Checkpoint). Die Konfiguration des Systems mit allen zuvor gesicherten Parametern stellt den Snapshot wieder her (Restore).

6.8.3. Skriptsprachen-API

Das zusammen mit Texas Instruments (TI) durchgeführte GreenSocs-Projekt GreenScript [Bart10][i] soll SystemC-Simulationen in verschiedene Scriptsprachen integrieren. Aktuell existiert eine Implementierung für Python. GreenScript beinhaltet unter anderem auch eine Schnittstelle zur Konfiguration der Modelle mit GreenConfig.

6.9. Performance-Betrachtung

Dieser Abschnitt diskutiert Performanceaspekte bezüglich der Ausführungsgeschwindigkeit, die für die in dieser Arbeit vorgestellten Werkzeuge relevant sind. Konkret betrifft das den universellen Konfigurationsmechanismus GreenConfig und die unterliegende Middleware GreenControl.

Identifizierung relevanter Aspekte

Für SystemC-Simulationen ist vor allem die Laufzeit der Simulationsphase interessant, da ein Großteil der gesamten Simulation im Wesentlichen auf die Simulationsphase entfällt, also nicht während des Aufbaus und der Initialisierung des Systems. Eine Übersicht der im Folgenden genannten Aspekte zeigt Tabelle 6.17. Sie nimmt auch eine Bewertung der Relevanz für die Konfiguration und Analyse vor.

Zunächst soll sichergestellt sein, dass die simple Existenz der Werkzeuge in der Simulation keinen negativen Einfluss auf die Simulationslaufzeit hat. Weiterhin sollen konkret GreenConfig-Parameter durch ihre Existenz – so lange sie nicht verwendet werden – keinen negativen Einfluss haben.

Sobald Konfiguration und Analyse während der Simulationsphase stattfinden sollen, sind als weitere Faktoren für die Performance manipulative und wertneutrale Zugriffe auf Modell-Parameter zu untersuchen.

Für die Gesamtsimulationslaufzeit ist die Modell-Konfiguration[28] und damit die Performance des Konfigurationsmechanismus zunächst vernachlässigbar: Die klassische Konfiguration findet vor der Simulationsphase statt, der Beitrag zur Simulationslaufzeit ist also gering. Das betrifft sowohl alle manipulativen wie auch alle wertneutralen Zugriffe. Die Modell-Steuerung muss dagegen genauer betrachtet werden. Sollen Variablen des Modells konfiguriert werden, auf die das Modell während der Simulationsphase lesend zugreift, kann die Konfiguration performancerelevant sein. Unter Einhaltung sehr einfacher Programmierregeln kann das aber verhindert werden, da Parameter mit Hilfe ihrer Callbacks bei Änderungen selbst aktiv werden können[29]. Es kann also immer verhindert werden, dass das Modell direkt auf Parametern arbeiten muss. Der Konfigurationsmechanismus hat aber eine hohe Benutzerfreundlichkeit zum Ziel, und es ist aus Sicht des Modellentwicklers wünschenswert, GreenConfig-Parameter direkt verwenden zu können. Deswegen lohnen sich Anstrengungen für eine hohe Performance.

Sobald GreenConfig-Parameter für die Analyse eingesetzt werden sollen, wird deren Performance relevant: Für die Modell-Untersuchung zu Analysezwecken müssen typischerweise Variablen gelesen werden (wertneutraler Zugriff), auf die das Modell während der Simulationsphase schreibend und lesend zugreift (manipulativer und wertneutraler Wertzugriff).

Zusammenfassend ist eine hohe Performance der Modell-Konfiguration für den Kernbeitrag dieser Arbeit grundsätzlich unbedeutend. Für die aus Benutzersicht einfache Verwendung von Parametern sowie für die Analyse ist eine hohe Performance während der Simulationsphase dennoch von Nutzen. Deswegen wurde bei der Entwicklung sowohl von GreenControl als auch des GreenConfig-Services großer Wert auf hohe Performance gelegt. Im Folgenden wird dies durch Messungen und Argumentation belegt und begründet.

Diskussion der relevanten Aspekte

Ein grundlegend wichtiger Aspekt ist, dass die Existenz der Middleware und der Services keinen negativen Einfluss auf die Simulationszeit haben soll. Die Middleware und der Konfigurations-Service werden instanziiert. Es handelt sich zunächst um eigenständig existierende

[28] Nach Abbildung 2.4 auf Seite 12 besteht die Modell-Konfiguration aus klassischer Konfiguration und Modell-Steuerung.

[29] Die Programmierregel könnte wie folgt aussehen: Der konfigurierbare Parameter wird als zusätzliche Variable im Modell eingeführt und ein post_write-Callback registriert. Bei einer Änderung, also Konfiguration von außerhalb des Modells, löst der Parameter einen Callback aus, der dann die unveränderte lokale Variable beschreibt. Damit wird ein negativer Einfluss auf die Performance nahezu ausgeschlossen.

Objekte ohne Verbindung zum simulierten Modell oder dem SystemC-Simulationskernel. Die für die Simulationslaufzeit als besonders relevant identifizierte Simulationsphase wird vom SystemC-Kernel kontrolliert, der das Scheduling durchführt und Aktionen im Modell ausführt. Da die Werkzeuge in der Ausgangssituation keinerlei Verbindung zum Kernel haben, werden sie während der Simulationsphase nie aufgerufen[30].

GreenConfig-Parameter werden typischerweise im Modell instanziiert, zum Beispiel als SystemC-Modul-Member. Solange keine Zugriffe stattfinden, sind sie als Klassenmember nahezu ohne Bezug zum SystemC-Kernel bzw. dessen Zeitablaufsteuerung. Sie sind lediglich SystemC-Objekte, deren Existenz aber ebenfalls keinen nennenswerten Einfluss auf die Performance der laufenden Simulation hat.

Sobald Zugriffe auf GreenConfig-Parameter während der Simulationsphase stattfinden, gibt es naturgemäß zusätzlichen Overhead, der im Folgenden untersucht werden soll. Die performance-kritischen Zugriffe sind die während der Simulationphase wiederholt ausgeführten Zugriffe. Sie können unabhängig vom Verursacher (von außen bzw. vom Modell) direkt auf dem Parameterobjekt stattfinden[31], unterscheiden sich also nicht. Deswegen ist die Herkunft der Zugriffe im Folgenden unerheblich.

Beim Design von GreenConfig wurde bei den potentiell performance-kritischen Funktionen Wert auf wenig Overhead gelegt. Vor allem betrifft das die manipulativen und wertneutralen Zugriffe über die Gleichheits- und Klammeroperatoren bzw. die entsprechenden Funktionen `setValue` und `getValue` der GreenConfig-Parameter.

Abbildung 6.18 zeigt in einem Aktivitätsdiagramm[32] die Aufrufstacks der beiden Zugriffsarten für einen Parameter, der keine registrierten Callbacks besitzt[33]. Die Funktion `operator=` (vgl. Abbildung 6.18(a)) führt einen manipulativen Wertzugriff durch, die Funktion `operator()` (vgl. Abbildung 6.18(b)) führt einen lesenden, wertneutralen Zugriff durch. Die in Abbildung 6.18(c) gezeigte Hilfsfunktion wird aus den Funktionen aufgerufen, die die verschiedenen Callbacks behandeln. Aus der Abbildung ist ersichtlich, dass für das Bereitstellen verschiedener Merkmale der im Folgenden gelistete Overhead notwendig ist[34].

- Für jeden Callback-Typ ist eine Prüfung auf die Existenz von Callbacks notwendig (`empty()` in Abbildung 6.18(c)). Der Overhead ist für alle Arten von Callbacks identisch.
- Die Auswertung findet in einer `if`-Anweisung statt.
- Um das Sperren von Parametern zu ermöglichen, ist beim manipulativen Zugriff eine `if`-Anweisung notwendig, die auf einen Wahrheitswert (Bool) prüft.

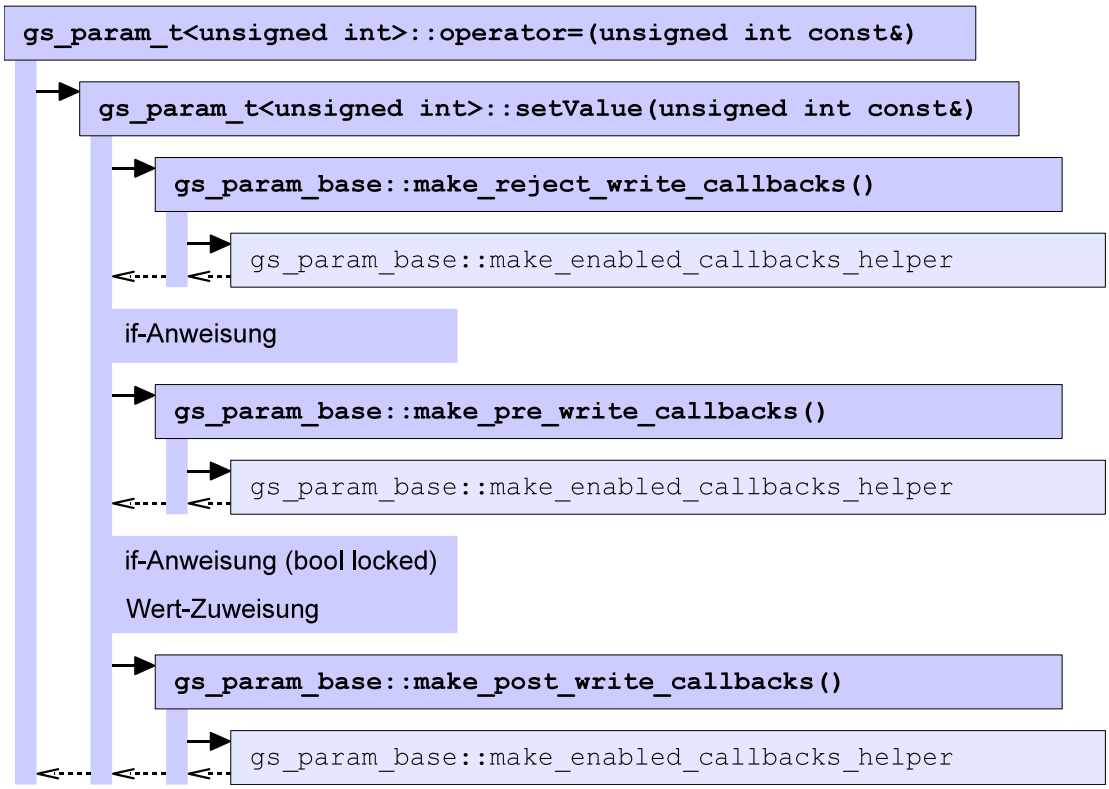
[30] GreenControl registriert sich beim Kernel für die Elaborations- und Simulations-Callbacks, die aber nicht während der Simulationsphase aufgerufen werden.

[31] Soll die Konfiguration oder Analyse eines Modells auf die Performance hin optimiert werden, ist es empfehlenswert und nahezu immer möglich, mit Referenzen oder Pointern direkt auf dem Parameterobjekt zu arbeiten. Deswegen werden diese Zugriffe hier untersucht und typunabhängige Zeichenkettenzugriffe nicht betrachtet.

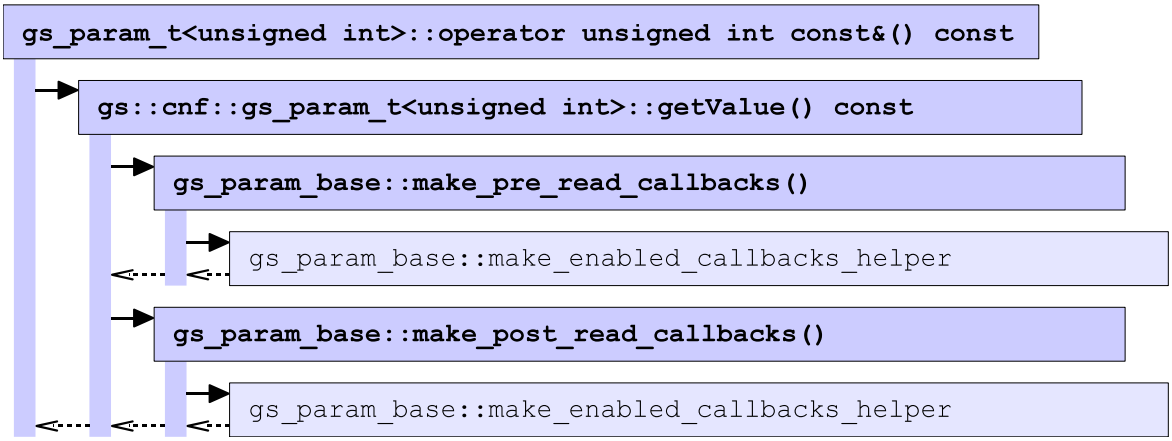
[32] Das Aktivitätsdiagramm ist erweitert um die dabei ausgeführten C++-Anweisungen.

[33] Sobald ein Parameter Callbacks besitzt, besteht konkretes Interesse am Parameterzugriff. Dadurch halte ich den dann zusätzlichen Overhead als ohnehin vom Benutzer oder Tool gewünscht oder akzeptiert.

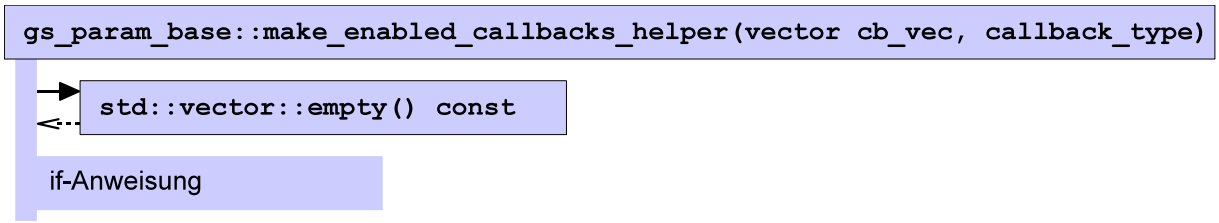
[34] Die eigentliche Wert-Zuweisung ist kein Overhead, da die Zuweisung auch ohne Parameter notwendig wäre.



(a) Aufrufstack Zuweisungsoperator



(b) Aufrufstack Klammeroperator



(c) Aufrufstack Hilfsfunktion

Abbildung 6.18.: Aufrufstacks der Parameterzugriffe; die in (c) dargestellte Hilfsfunktion wird in (a) und (b) jeweils wiederholt verwendet.

- Um reject_write-Callbacks zu ermöglichen, ist beim manipulativen Zugriff eine `if`-Anweisung notwendig, die die Rückgabe der reject_write-Callbacks auswertet.

Die Auswirkungen dieser Overheads sind den Messungen im nächsten Unterabschnitt zu entnehmen.

Messungen

Zwei verschiedene Messungen werden an einem Beispielsystem durchgeführt: Zum einen werden die zusätzlichen Instruktionen gezählt, die die Verwendung von GreenConfig-Parametern hervorrufen; zum anderen werden Laufzeitmessungen mit und ohne Parameter durchgeführt.

Das hier beschriebene Beispielsystem soll einen möglichst typischen Simulationsaufbau widerspiegeln. Das ist jedoch nur eingeschränkt möglich, da es sehr klein gehalten ist und kein reales Verhalten simuliert. Die Intensität der Verwendung von Parametern kann konfiguriert werden, sodass die Ergebnisse relativ zu realen Simulationen aussagekräftig oder sogar verschärft sind. Das Beispielsystem sowie alle Messergebnisse und deren Datengrundlagen können im Beispiel B.16 nachvollzogen werden.

Das *Beispielsystem* besteht aus einem Master und einem Slave, die über TLM-Ports direkt miteinander verbunden sind. Für die Kommunikation werden blockende GSGP-Sockets verwendet[35].

Der Master ist ein Traffic-Generator, der in einer Schleife eine per Makro festlegbare Anzahl von Schreib- und Lesetransaktionen schickt. Die geschriebenen Daten sind dabei jeweils der Wert des Schleifenzählers. Für die Messungen werden zwei Varianten verwendet: Die Laufzeitmessungen werden in einer „großen“ Simulation mit je 100 000 000 Write- und Read-Transaktionen durchgeführt. Die Instruktionszählungen werden in einer „kleinen“ Simulation mit je 1 000 000 Transaktionen durchgeführt, um die Laufzeit mit dem Tool Valgrind zu begrenzen.

Der Slave führt bei jeder eintreffenden Transaktion abhängig vom Befehl einen schreibenden oder lesenden Zugriff auf einen Klassenmember durch. Diese Datenvariable ist entweder ein GreenConfig-Parameter oder zum Vergleich ein primitiver oder SystemC-Datentyp (`gs_param<unsigned int>`, `unsigned int` oder `sc_uint<32>`). Optional wird anschließend mit einer Schleife eine Verhaltenssimulation nachempfunden: eine tausend Mal durchgeführte Berechnung des Quadrats einer `volatile`-Variablen[36].

Für jede Messung ergeben sich je 100 000 000 bzw. 1 000 000 schreibende und lesende Parameterzugriffe nach Abbildung 6.18.

Die Simulation wird auf einem MacBook Pro[37] mit gcc 4.0.1 und der Optimierung -O3 kompiliert und ausgeführt.

[35] GSGP-Sockets sind GreenSockets und werden im GreenSocs-Projekt unter `http://www.greensocs.com/projects/GreenSocket/GSGPSocket` zur Verfügung gestellt.
[36] Wenn nicht anders angegeben, ist die Verhaltenssimulation in den Messungen aktiviert.
[37] Prozessor 2,4 GHz Intel Core 2 Duo, 4 GB RAM, Betriebssystem Mac OS X 10.5.8 (Leopard)

	Datentyp	unsigned int	sc_uint<32>	gs_param<unsigned int>
(1)	Laufzeit	04m10,88s	04m12,25s	04m09,22s
	Vergleich	100%	100,5%	99,4%
(2)	Laufzeit	29,54s	29,91s	30,56s
	Vergleich	100%	97,9%	103,4%

Tabelle 6.19.: Performance-Messung; (1) mit, (2) ohne Verhaltenssimulation

Um die *Anzahl der zusätzlichen Instruktionen* bei der Verwendung von GreenConfig-Parametern zu erhalten, wird die „kleine“ Simulation mit der Werkzeugsammlung Valgrind und dem Tool Callgrind[38] ausgeführt. Das Tool erstellt umfangreiche Statistiken, unter anderem einen Callstack und Zählungen der jeweils benötigten Instruktionen.

Die Auswertung[39] ergibt insgesamt 40 000 000 zusätzliche Instruktionen. Das sind für jeden Parameterzugriff (Lese- oder Schreibtransaktion) 40 Instruktionen. Bezogen auf die Gesamtanzahl von 13 215 166 710 Instruktionen der Simulation (gemessen im SystemC-Thread `ExplMaster2::run()`) sind das 0,30%. Bezogen auf die Anzahl der Instruktionen der Slave-Funktion `IPmodel` (12 354 000 000, 93,48% der Gesamtanzahl) sind das 0,32%.

Für die Ermittlung des *Laufzeit-Overheads* wird die Laufzeit der „großen“ Simulation innerhalb der Anwendung gemessen[40]. Dabei wird die Zeit gemessen, die der Aufruf von `sc_start()` benötigt. Tabelle 6.19 zeigt Ergebnisse verschiedener Varianten, jeweils Mittelwerte aus mehreren Durchgängen[41].

Tabelle 6.19 zeigt in Zeile (1) zunächst die Ergebnisse der oben beschriebenen Simulation: Die Variante mit der Datenvariable als primitiver Datentyp `unsigned int` wird als Referenz (100% Laufzeit) verwendet. Im Vergleich mit dieser Referenz sind die Laufzeiten der beiden Simulationen mit einem SystemC-Datentyp und dem Parameter im Rahmen der Messgenauigkeit identisch, obwohl diese tatsächlich aufwändiger sind. Die geringen Unterschiede, die sogar zugunsten der Messung mit Parametern ausfallen, können zum Beispiel durch Caching-Effekte hervorgerufen werden. Das bedeutet, dass in einer Simulation mit relativ wenigen Parameterzugriffen kein messbarer negativer Einfluss auf die Simulationslaufzeit festgestellt wird.

Der Aufwand der Simulation lässt sich fast beliebig verringern, sodass der Anteil der Parameterzugriffe an der Laufzeit größer wird und schließlich der Overhead der Parameterzugriffe doch messbar wird: Wird die Schleife der Verhaltenssimulation im Slave weggelassen, lässt sich ein Laufzeit-Overhead von 3,4% feststellen, vgl. Tabelle 6.19, Zeile (2). Callgrind-Ergebnisse für diese Variante ergeben einen Anteil der Instruktionen für die Parameterzugriffe von 3,27% an der Gesamtanzahl und 11,05% an der Slave-Funktion. Das bedeutet selbst in

[38] Valgrind und Callgrind siehe `http://valgrind.org/`.

[39] Die Callgrind-Ergebnisse sind im Beispiel B.16 in den Dateien callgrind.out.* gespeichert und in der Datei callgrind_Ergebnisse_volatile.txt ausgewertet.

[40] Alle Messungen werden im Single-User-Mode mit hoher Priorität durchgeführt, um Beeinträchtigungen durch andere Prozesse zu minimieren.

[41] Die vollständigen Messdaten sind im Beispiel B.16 in der Datei performance_results.xls gelistet.

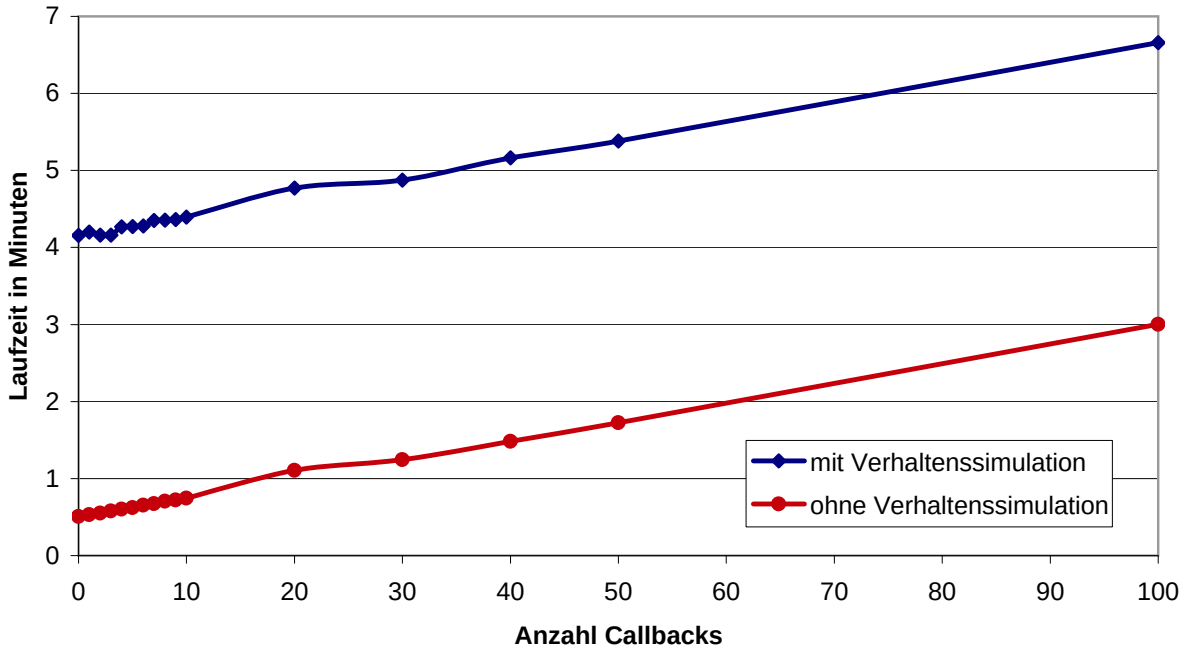


Abbildung 6.20.: Laufzeitmessungen mit verschiedener Callback-Anzahl

einer Simulation, in der extrem viele Parameterzugriffe getätigt werden, ist der Overhead verhältnismäßig niedrig.

Messungen mit Callbacks

Ein weiterer Performance-Aspekt sind die Kosten von Callbacks. Wenn bei einem Parameter vom Benutzer bzw. dem Tool Callbacks registriert werden, sind Performance-Einbußen akzeptabel und werden zusätzlich durch die Anweisungen des Benutzers bzw. Tools in den Callback-Funktionen verstärkt. Trotzdem ist es interessant, welchen Overhead registrierte Callbacks bedeuten und wie dieser skaliert.

Dazu wird das oben vorgestellte Beispielsystem für entsprechende Messungen erweitert: Ein Beobachter-Objekt registriert eine konfigurierbare Anzahl Callbacks auf dem GreenConfig-Parameter im Slave. Da sich der Overhead aller Callbacks wie bereits diskutiert nur minimal unterscheidet, werden hier reject_write-Callbacks registriert, die durch die zusätzliche `if`-Anweisung den größten Overhead haben. Die Callback-Funktionen selbst sind leer[42], es werden also nahezu ausschließlich die Zusatzkosten für den Aufruf der Callbacks gemessen. Abbildung 6.20 zeigt zwei Messreihen: mit und ohne Verhaltenssimulation[43]. Während die Anzahl der registrierten Callbacks steigt, zeigt sich ein linearer Anstieg der Laufzeit. Das bedeutet, jeder zusätzliche Callback ist immer gleich teuer. Als Referenz beginnen beide Messreihen mit der Messung ohne Callbacks aus Tabelle 6.19 – hier zeigt sich kein Laufzeitsprung; das Vorhandensein von Callbacks hat folglich keinen über den Effekt des Aufrufs

[42] Damit die Funktionsaufrufe vom Compiler im Zuge der Optimierung nicht vollständig entfernt werden, findet in der Callback-Funktion ein Zugriff auf einen `volatile` Member statt.
[43] Die vollständigen Messdaten sind im Beispiel B.16 in der Datei performance_results.xls gelistet.

hinausgehenden negativen Einfluss auf die Laufzeit. Bei einer sinnvollen Anzahl von Callbacks (unter zehn) ist der Effekt gering.

Zusammenfassung

Für die Modell-Konfiguration ist eine hohe Performance von GreenConfig-Parametern nicht zwingend notwendig, aber hilfreich. Für die Analyse ist ihre Performance dagegen wichtig. Die als relevant identifizierten Aspekte von GreenConfig-Parametern sind wertneutrale und manipulative Wertzugriffe auf das Parameterobjekt. Der Overhead der betreffenden Funktionen ist mit sehr wenigen zusätzlich benötigten Anweisungen gering. Eine moderate Häufigkeit von Parameterzugriffen hat keinen messbaren Einfluss auf die Simulationslaufzeit, erst intensive Verwendung macht sich mit geringen, aber absolut akzeptablen Einbußen bemerkbar.

7. Zusammenfassung und Ausblick

Inhalt

Der Entwurf komplexer Hardware-Software-Systeme wird aktuell unter Verwendung unterschiedlich abstrakter Modelle bewältigt. Dabei kommen Systembeschreibungssprachen wie SystemC in den Entwicklungsumgebungen der verschiedenen Tool-Hersteller zum Einsatz. Die Wiederverwendbarkeit der Modelle zwischen verschiedenen Projekten sowie der Austausch der Modelle über Firmengrenzen hinweg ist wichtig für die Produktivität. Durch die verschiedenen proprietären Entwicklungsumgebungen sind die Modelle jedoch nur eingeschränkt interoperabel, da nur die Real-Interoperabilität der Modelle selbst durch TLM-2.0 standardisiert ist. Dagegen ist die Konfigurierbarkeit der Modelle in jeder Entwicklungsumgebung anders realisiert, womit eine Meta-Interoperabilität der Modelle normalerweise fehlt.

In der vorliegenden Arbeit wurde das Framework GreenConfig für die Konfiguration von Hardware-Software-Modellen in SystemC erarbeitet. Es ermöglicht die Austauschbarkeit von Modellen zwischen verschiedenen Entwicklungsumgebungen und erhöht somit deren Wiederverwendbarkeit. Aufbauend auf der Modell-Middleware GreenControl bietet der flexibel erweiterbare Konfigurationsmechanismus alle Merkmale der im Stand der Technik gefundenen Mechanismen, die im Folgenden aufgegriffen wurden. Zudem bietet GreenConfig Adapter zu diesen fremden Mechanismen, was die Austauschbarkeit der Modelle bezüglich des Konfigurationsaspekts ermöglicht. Die Erkenntnisse sowie das Framework selbst sind in der Arbeitsgruppe OSCI CCI in die Standardisierungsarbeiten zur Konfigurierbarkeit eingeflossen. Zusätzlich ermöglicht GreenConfig eine Modell-Untersuchung, beispielsweise für die Analyse.

7.1. Zusammenfassung und Ergebnisse

Nach einer Einleitung wurden in Kapitel 2 zunächst die Grundlagen für die Konfiguration und Untersuchung von Hardware-Software-Modellen in SystemC gelegt: Die Meta-Interoperabilität grenzt die Werkzeugebene von der Real-Interoperabilität der Modelle selbst ab. Der Stand der Technik wurde mit einem Überblick über existierende akademische und industrielle Konfigurationsmechanismen behandelt.

Diese Arbeit hat sich auf die Konfigurations-Interoperabilität als Spezialfall der Meta-Interoperabilität konzentriert. Kapitel 3 hat verschiedene der zuvor untersuchten Mechanismen analysiert und daraus verschiedene Ergebnisse und Definitionen abgeleitet: Allen Mechanismen gemeinsam ist der Modell-Parameter als die zu konfigurierende Eigenschaft. Unterschiede ergaben sich in den verschiedenen Konfigurationsansätzen des Konfigurations-Interface und des Class-Wrapper), von denen letzterer als mächtiger identifiziert wurde. Deswegen verwendet mein universeller Konfigurationsmechanismus den Class-Wrapper-Ansatz. Ein weiterer Unterschied bestand in den zeitlichen oder hierarchischen Konfigurationsrangfolgen, von denen die zeitliche weiter verbreitet ist und folglich vom hier entwickelten Mechanismus verwendet wird. Sie ermöglicht jedoch die Nachbildung der hierarchischen Rangfolge.

Aus dem Stand der Technik und der darauf basierenden Analyse ergaben sich in Kapitel 3 Anforderungen an den universellen Mechanismus. Er sollte in der Lage sein, die Konfigurations-Interoperabilität über Adapter herzustellen. Die Integration „Proprietäres Modell in universeller Entwicklungsumgebung (PIU)“ ermöglicht es, ein Modell aus einem fremden Konfigurationsmechanismus in einer Umgebung mit dem universellen Mechanismus zu konfigurieren. Die Integration „Universelles Modell in proprietärer Entwicklungsumgebung (UIP)“ macht dagegen ein Modell unter dem universellen Mechanismus in proprietären Umgebungen konfigurierbar, etwa in kommerziellen Tools wie dem Synopsys Innovator. Mischt man diese Verfahren, lassen sich teilweise sogar verschiedene Mechanismen verwendende Modelle in eine proprietäre Entwicklungsumgebung einbinden. Dies wurde später in Abschnitt 6.6 für den CoWare Platform-Architect demonstriert.

Meine Erkenntnisse sind in die Standardisierungsaktivitäten der Arbeitsgruppe OSCI CCI zur Konfigurierbarkeit eingeflossen, was die Aktualität und industrielle Relevanz dieser Arbeit unterstreicht. Der zu diesem Zeitpunkt dort verwendete Prototyp wurde von mir entwickelt und setzt intern den GreenConfig-Konfigurationsmechanismus ein, weshalb er den industriellen Anforderungen genügt.

Im Kapitel 4 wurde die Modell-Middleware GreenControl erarbeitet. Sie stellt dem Konfigurationsmechanismus eine flexible Kommunikation zur Verfügung. Middleware-Services wie die Konfiguration werden in mehrere Elemente zerlegt: Das Service-Plug-in stellt die zentrale Funktionalität bereit, und mehrere User-APIs ermöglichen verschiedene Arten des Zugriffs. Die Elemente der Middleware kommunizieren über den zentralen Core. Dafür werden erweiterbare Transaktionen verwendet, die für den jeweiligen Service optimierte Daten effizient übertragen können.

Die Middleware wurde als unterliegende Kommunikation für den universellen Konfigurationsmechanismus entworfen, kann aber auch als Plattform für weitere Services verwendet werden, die der Meta-Interoperabilität dienen. Ein solches Beispiel ist der Analyse-Service aus Anhang A. Für die industrielle Praxis kann insbesondere die Binärkompatibilität verschiedener User-API-Versionen nützlich sein: Die Middleware ermöglicht das Verbinden von binären, vorkompilierten Modellen, deren User-APIs aus unterschiedlichen Versionen des

Konfigurationsmechanismus stammen können, solange das Plug-in das neuste ist bzw. mit den Transaktionen aller angeschlossenen User-APIs umgehen kann.

Der universelle Konfigurationsmechanismus GreenConfig wurde in Kapitel 5 vorgestellt. Er ist als Service der Modell-Middleware realisiert und erbt somit dessen Flexibilität für die interne API zwischen den User-APIs und der zentralen Funktionalität im Plug-in. Das zentrale Konfigurations-Plug-in verwaltet eine Datenbank aller Modell-Parameter der Simulation. Es besitzt eine optional verwendbare Schnittstelle für Adapter zu anderen Parameter-Datenbanken, beispielsweise der eines proprietären Mechanismus wie CoWare SCML.

Die in GreenConfig verwendeten Modell-Parameter `gs_param` sind Class-Wrapper-Objekte. Sie werden entweder im Modell als konfigurierbares Element (Modell-API) verwendet und ersetzen dort transparent ein Datenobjekt, oder sie werden innerhalb eines Adapters verwendet, um zum Beispiel einen Modell-Parameter eines fremden Mechanismus zu spiegeln. Diese Parameter können vom Entwickler eines portablen Modells verwendet werden, denn verschiedene Adapter exportieren diese Parameter zu proprietären Konfigurationsmechanismen.

Eine Vielzahl von Merkmalen stellt die Integrationsfähigkeit von GreenConfig sicher, die ein Alleinstellungsmerkmal unter den existierenden Konfigurationsmechanismen ist. Ein solches Merkmal ist die Laufzeitkonfiguration, wo manipulative Wertzugriffe während der Simulationsphase möglich sind. Ein weiteres bedeutsames Merkmal sind verschiedene Callback-Typen, die umfassenden Zugriff auf die GreenConfig-Parameter bei verschiedenen Ereignissen ermöglichen. Dadurch können alle Aktionen auf Parametern vor oder nach ihrer Ausführung abgegriffen werden, was zum Beispiel das Spiegeln von Modell-Parametern zu oder von proprietären Parametern ermöglicht.

Die GreenConfig-API wurde als zentrale Tool-API von GreenConfig für die Konfiguration von Modellen eingeführt. Sie ermöglicht unter anderem das Setzen und Lesen von impliziten Parametern vor der Existenz eines Parameterobjekts und den Zugriff auf alle expliziten Parameter, das heißt Parameterobjekte beliebigen Typs in der Simulation.

Die umfassenden Merkmale der GreenConfig-Parameter und der GreenConfig-API ermöglichten den Entwurf von Adapter-APIs. Diese unterstützen entweder das Verbinden von Modellen mit proprietären Konfigurationsmechanismen zum universellen Mechanismus GreenConfig (PIU-Integration), oder die Adapter-APIs ermöglichen das Anbinden eines die GreenConfig-Parameter verwendenden Modells oder eines kompletten universellen Systems in eine proprietäre Entwicklungsumgebung (UIP-Integration). Der Umfang der UIP-Integration ist eingeschränkt durch die Fähigkeiten des Zielmechanismus. Im Abschnitt 5.9 wurden allgemeine Ansätze für die Integrationen vorgestellt, die im anschließenden Kapitel für kommerzielle Mechanismen konkretisiert wurden.

In Kapitel 6 wurden zunächst verschiedene Adapter-APIs für wesentliche industrielle Konfigurationsmechanismen vorgestellt, nämlich SCML im Platform-Architect, CCSS-Parameter im Synopsys Innovator, ARM CASI und OVM. Außerdem wurde die Nachbildung hierarchischer Rangfolge in der zeitlichen Rangfolge im Detail betrachtet. Als Proof-of-Concept

wurde anschließend ein großes Integrationsbeispiel entwickelt, das verschiedene Modelle innerhalb der Entwicklungsumgebung Platform-Architect integriert, von denen jedes einen der zuvor beschriebenen Adapter verwendet, also für Konfigurationsmechanismen anderer Hersteller entwickelt wurde. Im Anschluss daran wurden weitere Anwendungsbeispiele präsentiert: SCRSI als grafisches Tool für verschiedene GreenControl- und GreenConfig-Funktionen und die Simulationskontrolle, entstanden in mehreren studentischen Arbeiten. Danach wurden weitere akademische und industrielle Projekte kurz vorgestellt, die GreenConfig verwenden. Abschließend wurde die gute Performance des universellen Konfigurationsmechanismus gezeigt.

Fazit

Als konkreter Erfolg dieser Arbeit kann beispielsweise der in der Einleitung erwähnte Zulieferer für Mobiltelefonhersteller nun die Modelle der Peripherie mit einem universellen Mechanismus einheitlich konfigurierbar machen. Eine Anpassung an verschiedene Entwicklungsumgebungen unterschiedlicher Kunden für unterschiedliche virtuelle Prototypen ist nicht mehr notwendig.

Der in dieser Arbeit entstandene flexible Konfigurationsmechanismus unterstützt die Konfigurations-Interoperabilität von SystemC-Modellen aus unterschiedlichen Entwicklungsumgebungen. Entweder sind Modelle nun unabhängig von Tools konfigurierbar, oder es werden bestehende Modelle durch Adapter einem fremden Konfigurationsmechanismus verfügbar gemacht.

Ein solcher praxisrelevanter Ansatz ist bisher in der Forschung meines Wissens nicht verfolgt worden. Ein universeller und von Entwicklungsumgebungen unabhängiger Konfigurationsmechanismus ist für jene Modell-Entwickler von Interesse, die sich nicht an ein kommerzielles Herstellertool binden möchten. In der Industrie zeigen zum Beispiel die Firmen Intel, Texas Instruments und die European Space Agency (ESA) ein solches Interesse.

Zudem sind die Untersuchungen dieser Arbeit und der universelle Mechanismus auch für die Hersteller von SystemC-Entwicklungsumgebungen bedeutsam, sofern sie Interesse an portablen Modellen haben. Die Anstrengungen der OSCI zur Standardisierung der Konfiguration belegen, dass dieses Interesse bei Tool-Herstellern und deren Kunden tatsächlich besteht.

7.2. Ausblick

Das Konfigurationsframework GreenConfig und die Modell-Middleware GreenControl sind als Open-Source-Projekt auf der GreenSocs-Webseite verfügbar. Das Projekt wird von einigen akademischen Projekten und der Industrie verwendet. Es kann bei neuen Anforderungen weiterentwickelt werden. Ebenso können weitere Middleware-Services hinzugefügt werden.

Die Arbeit der Arbeitsgruppe OSCI CCI am Konfigurationstandard ist noch nicht beendet. Es erscheint sinnvoll, GreenConfig auf zwei Arten an den kommenden OSCI-Konfigurations-

standard anzupassen: Zum einen ist es erstrebenswert, die GreenConfig-Parameter verwendenden Modelle ohne Code-Änderungen am Modell standardkonform werden zu lassen, zum anderen sollte GreenConfig aktuell gehalten werden, sodass es weiterhin eine Implementierung des Standards bleibt. Auf diese Weise bliebe eine freie Implementierung des Standards verfügbar.

Über die Konfigurations-Interoperabilität hinaus beinhaltet die Meta-Interoperabilität noch folgende weiteren Aspekte, die für das Zusammenspiel zwischen Modellen und den Entwicklungsumgebungen wichtig sind.

Die Analyse und das Monitoring von Vorgängen innerhalb von Simulationen kann über diese Arbeit hinaus untersucht und standardisiert werden. Beispielsweise definiert der Standard TLM-2.0 hierzu das Debug-Transport-Interface, das einen Anfang in diese Richtung darstellt.

Der Austausch von Meta-Nachrichten zwischen Modell und Benutzer oder anderen Modellen, wie er in Ansätzen vom GreenSocs-Projekt GreenMessage behandelt wird, kann für die Steuerung der Simulation ebenfalls untersucht und standardisiert werden. Solche Nachrichten könnten mit einer einheitlichen Semantik Stimulationen wie eine Fehlerinjektion in das Modell bringen. Möglicherweise wird derartiges dann Inhalt des CCI-Standards.

Ein weiterer großer Bereich der Meta-Interoperabilität ist das Debugging. Darunter sind sowohl Verfahren für die Fehlersuche im Hardware-Teil des Modells zu verstehen als auch das Debuggen von Software, die auf simulierter Hardware ausgeführt wird. Wünschenswert sind einheitliche Interfaces für den Debug-Vorgang selbst sowie Richtlinien und Standards für das Ausrüsten von Modellen mit Debugging unterstützenden Fähigkeiten.

Um die Komplexität der Standards gering zu halten, sollten jedoch stets die Kosten und der Nutzen abgewogen werden, wo sich eine Standardisierung lohnt und welche Aspekte dagegen sinnvollerweise lieber als Merkmale proprietärer Software realisiert werden sollten.

Anhänge

A. Analyse-Service

Inhalt

Neben dem Hauptbeitrag dieser Arbeit, dem Konfigurationsmechanismus, wird in diesem Kapitel ein Analyse-Service für die GreenControl-Middleware grob vorgestellt. Dieser zusätzliche Service demonstriert die Verwendung der Middleware für andere Zwecke als die Konfiguration. Zudem kann er ebenfalls dem Hauptziel dieser Arbeit, der Meta-Interoperabilität, dienen.

Der **Analyse-Service GreenAV** steht als Open-Source-Projekt als Teil vom GreenControl-Projekt [Schr11][i] bereit. AV steht hier für „Analysis and Visibility“. Der Service soll dem Systementwickler die Analyse des Systems und die Ausgabe bzw. Vorbereitung der Ergebnisse für die Visualisierung erlauben. Da sich die GreenConfig-Parameter, wie in Abschnitt 5.10 festgestellt, für die Modell-Untersuchung eignen, werden sie von GreenAV als Informationsquelle für die Analyse verwendet. Der Entwickler setzt an den zu untersuchenden Stellen Parameter ein, die anschließend automatisch analysierbar sind.

Eine Kernfunktion dieses Services ist die Möglichkeit, beliebige Parameteränderungen zu protokollieren, mit Formeln und statistischen Berechnungen zu analysieren und an verschiedene Ausgabemethoden für die Visualisierung weiterzuleiten.

Der sogenannte Statistik-Kalkulator ermöglicht eine konditionale Analyse von Parameterwerten. Mehrere Parameter können als Eingabewerte einer Formel verwendet werden, die das Ergebnis wiederum für die Weiterverarbeitung oder die Visualisierung als Parameter zur Verfügung stellt.

Verschiedene Ausgabemethoden sind modular über sogenannte Output-Plug-ins realisiert. Ausgaben sind beispielsweise einfache Text- oder Excel-Dateien, Konsolenausgaben oder spezielle Streams (SystemC Verification Standard (SCV) [OSCI03][i]), die von anderen (auch kommerziellen) SystemC-Analyse-Werkzeugen empfangen und weiterverarbeitet werden können. Damit werden die analysierten Daten für die Visualisierung vorbereitet – die tatsächliche

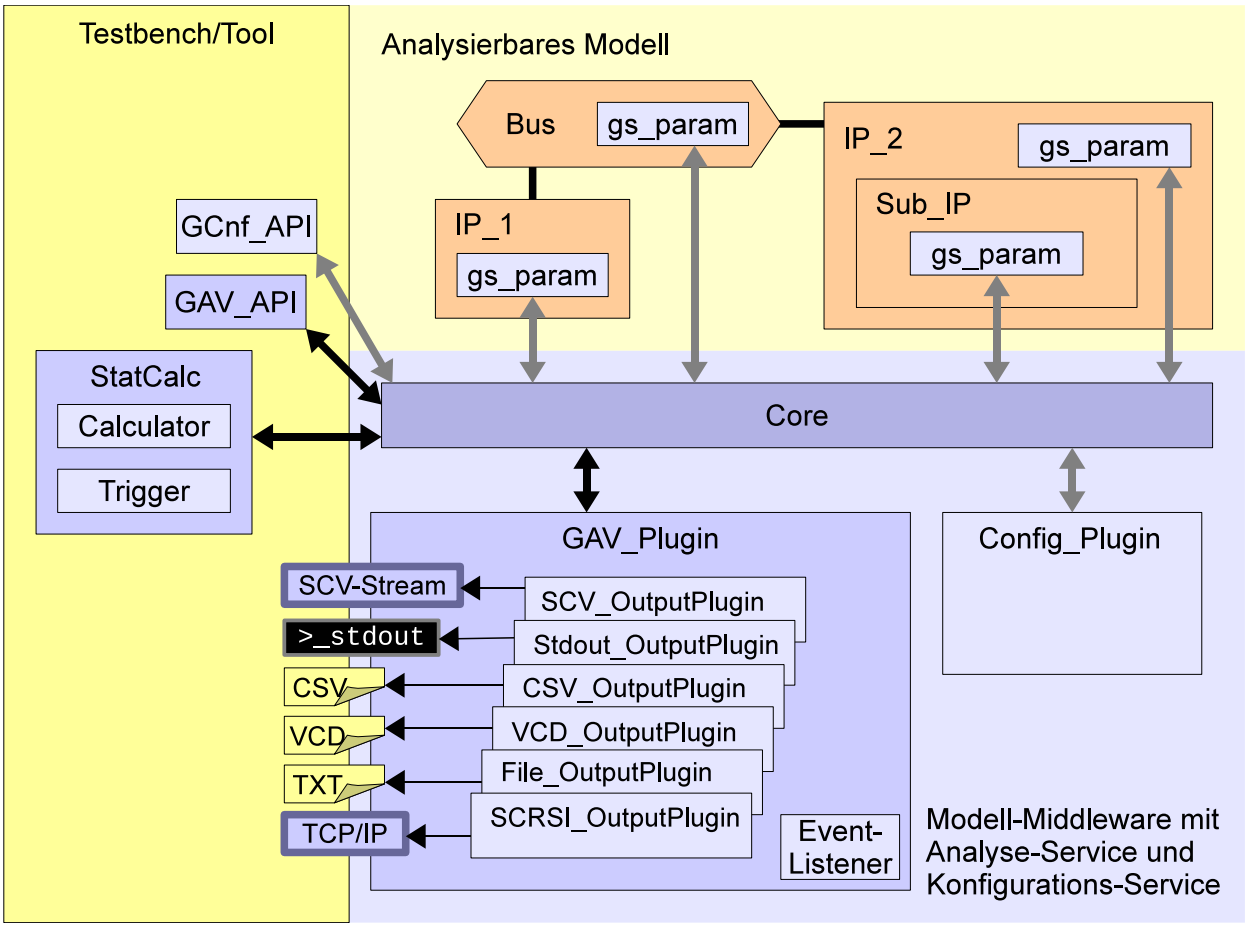


Abbildung A.1.: Grundlegende Struktur des Analyse-Services in einer Simulation (Beispiel).

grafische Visualisierung wird auf externe Tools verlagert. Solche Tools sind das eigene SCRSI aus Abschnitt 6.7 oder einige kommerzielle im Abschnitt A.4 gezeigte.

Zusätzlich zum in den folgenden Abschnitten beschriebenen Aufbau sind weitere Details und Benutzerhinweise zum Analyse-Service GreenAV in [ScKG10]^i zu finden.

A.1. Grundlegende Struktur

Dieser Abschnitt beschreibt die grundlegende Struktur des Analyse-Services GreenAV. Eine Übersicht zeigt Abbildung A.1. Interessante Details werden in den weiteren Abschnitten aufgegriffen.

Zunächst gibt die Modell-Middleware (vgl. Kapitel 4) den Aufbau mit einem GreenControl-Service-Plug-in und GreenControl-User-APIs vor. Die Verwendung von GreenAV setzt zwingend den Konfigurationsmechanismus GreenConfig voraus, da dessen Modell-Parameter die Grundlage für die Analyse darstellen.

A.1.1. Übersicht Analyse-Plug-in

Das **Analyse-Plug-in** (Klasse `GAV_Plugin`) verwaltet zentralisiert sämtliche Ausgabemöglichkeiten und ermöglicht den User-APIs den Zugriff darauf. Zusätzlich verwaltet es serviceintern verwendete Event-Listener. Es ist das Middleware-Service-Plug-in für den Analyse-Service und somit ein Singleton-Objekt (vgl. Abschnitt 4.1 auf Seite 70).

Output-Plug-ins

Ein **Output-Plug-in** ist eine Ausgabekomponente für GreenConfig-Parameter und bereitet damit GreenAV-Analyseergebnisse für die Visualisierung vor[1]. Verschiedene Typen von Output-Plug-ins arbeiten die Ergebnisse auf unterschiedliche Weise auf, um sie beispielsweise in Dateien zu exportieren. Das Analyse-Plug-in verwaltet in verschiedenen Listen zwei Arten von Output-Plug-ins: Default-Output-Plug-ins existieren einmal für jeden Typ. Wenn ein Benutzer für die Ausgabe eines Parameters nur den Output-Plug-in-Typ angibt, wird das entsprechende Default-Output-Plug-in verwendet. Zusätzlich gibt es eine Liste, in die beliebig viele Instanzen gleicher oder verschiedener Output-Plug-ins eingefügt werden können, denen Parameter zur Ausgabe übergeben werden können. Abbildung A.1 zeigt die verfügbaren Output-Plug-ins. In der Abbildung wird angedeutet, dass die verschiedenen Ausgaben vom Tool verwertet, also beispielsweise visualisiert werden können.

Das Analyse-Plug-in besitzt eine Factory, die unter Angabe des Typs eine entsprechende Output-Plug-in-Instanz erzeugt. Die Instanzen werden in den Listen gespeichert und auf Anforderung verwendet oder an eine User-API zurückgegeben.

Beim Erzeugen des Analyse-Plug-ins (z.B. in der Testbench) kann ein globales Standard-Output-Plug-in angegeben werden, das automatisch verwendet wird, wenn der Benutzer kein anderes auswählt.

Event-Listener

Event-Listener sind interne Hilfsstrukturen. Ähnlich dem Callback-Dispatcher der Modell-Middleware (siehe Abschnitt 4.2.4 auf Seite 82) minimieren Event-Listener die SystemC-Sichtbarkeit und erfüllen damit die Anforderung KA 15 auf Seite 46[2]. Event-Listener sind SystemC-Objekte (`sc_object`). Auf Anfrage des Benutzers (siehe Abschnitt A.1.2) erzeugen sie für ein übergebenes SystemC-Event einen darauf sensitiven Prozess. Wird das Event ausgelöst, benachrichtigen sie den Benutzer per Callback.

A.1.2. Übersicht GreenAV-API

Die **GreenAV-API** ist eine User-API für den Analyse-Service (Klasse `GAV_Api`). Über die GreenAV-API erhält der Benutzer Zugriff auf alle Output-Plug-ins. Er kann GreenConfig-

[1] Analyseergebnisse werden als GreenConfig-Parameter zur Verfügung gestellt.

[2] Wie der Callback-Dispatcher sind die Event-Listener ein zentraler Punkt, der kernelabhängig ersetzt werden kann, um die SystemC-Sichtbarkeit komplett zu unterbinden.

Parameter zur Beobachtung über die GreenAV-API oder direkt an die Output-Plug-ins übergeben und kann neue Output-Plug-ins erzeugen lassen.

A.1.3. Übersicht Statistik-Kalkulator

Der **Statistik-Kalkulator** (Klasse `StatCalc`) ist eine User-API des Analyse-Services und ermöglicht umfangreiche Analysen. Statistik-Kalkulatoren können vom Benutzer instanziiert werden, in Abbildung A.1 wird das beispielsweise vom Tool gemacht. Sie ermöglichen die Verrechnung mehrerer Eingabedaten (Modell-Parameter) zu einem Ausgabedatum (neu erstellter Modell-Parameter) unter Verwendung einer vom Benutzer konfigurierbaren Formel oder Statistikfunktion. Die Berechnung kann unter verschiedenen vom Benutzer auswählbaren Bedingungen ausgelöst werden.

Ein Statistik-Kalkulator besteht aus einem Kalkulator und einem Trigger:

Kalkulator

Der Kalkulator (Klasse `Calculator`) führt die tatsächliche Berechnung durch. Er wird vom Benutzer erzeugt, konfiguriert und anschließend dem Statistik-Kalkulator übergeben. Er führt auf Anforderung die zuvor definierte Berechnung durch und schreibt das Ergebnis in einen Ausgabeparameter. Die Berechnungen können aus mathematischen und logischen Operationen sowie einem Sliding Window kombiniert werden. Weitere Berechnungen können vom Benutzer definiert werden.

Trigger

Der Trigger (Klasse `Trigger`) übernimmt im Statistik-Kalkulator die Aktivierung einer Kalkulator-Berechnug. Das Standardverhalten ist die Aktivierung einer Neuberechnung nach jeder Änderung eines der Eingabeparameter; diesen Standard-Trigger erstellt der Statistik-Kalkulator eigenständig. Sollen andere Aktivierungsbedingungen gelten, kann ein Trigger vom Benutzer erzeugt, konfiguriert und dem Statistik-Kalkulator übergeben werden. Mögliche Aktivierungsbedingungen sind beispielsweise ein Wächter, der eine logische Bedingung prüft oder komplex berechnet, ein SystemC-Event oder manuelle Aktivierung.

Intern verwendet der Trigger eine GreenAV-API, um Zugriff auf den Event-Listener zu erlangen und eine GreenConfig-API für den Zugriff auf Eingabeparameter.

Ein Statistik-Kalkulator bzw. Trigger kann deaktiviert werden, wodurch der Performance-Overhead minimiert wird.

A.2. Anwendungsbeispiele

Ein umfangreiches Anwendungsbeispiel ist im externen Listing B.3 und in [ScKG08]^i ausführlich beschrieben. Dort werden verschiedene von GreenAV zur Verfügung gestellte Analysemechanismen anhand einer über einen Bus kommunizierenden Plattform demonstriert.

Beispielsweise zeichnen GreenConfig-Parameter im Cache-Modell die Hit- und Missraten mit einer Kalkulatorberechnung auf. Der verwendete Bus ist ebenfalls mit GreenConfig-Parametern ausgestattet, sodass Statistiken über die übertragenen Transaktionen pro Sekunde und deren Latenz erstellt werden können.

A.3. Merkmale

Dieser Abschnitt gibt eine Übersicht über die Merkmale, die vom Analyse-Service unterstützt werden. Details können [ScKG10][i] entnommen werden.

Allgemeine Merkmale

- Erfassen von GreenConfig-Parametern für die Ausgabe auf vielfältige Weise mit Hilfe von erweiterbaren Output-Plug-ins, z.B.
  - die Ausgabe in verschiedene Arten von Dateien und damit Erfüllung von Anforderung KA5 (siehe Seite 44),
  - das Aufzeichnen der Werte,
  - die Aufbereitung für die Visualisierung durch externe Tools,
- Berechnungen mit Parametern,
- statistische Auswertung von Parametern,
- bedingte Aktivierung von Berechnungen
  - bei Parameteränderungen,
  - SystemC-Event-basiert,
  - manuell nach Funktionsaufruf,
  - mit boolscher Vorbedingung,
  - einer Kombination der vorherigen,
- niedrige Performance-Einbußen durch Verwendung von GreenConfig-Parametern (vgl. Kapitel 6.9 und [MES+10]),
- bei deaktivierter Analyse keine zusätzliche Performance-Einbuße[3].

Besondere Merkmale

Das Analyse-Plug-in wird vom Benutzer instanziiert, um den Service verfügbar zu machen. Als besonderes Merkmal ist es möglich, die Instanziierung zu unterlassen, obwohl der Service vom Modell als existent erwartet wird. Als Folge steht der Service nicht zur Verfügung. Die Simulation läuft dennoch unbehindert, lediglich die Ausgabe der Output-Plug-ins unterbleibt.

[3] Der einzige Performance-Einfluss entsteht durch die Existenz von GreenConfig-Parametern ohne Callbacks, vgl. Kapitel 6.9.

Das demonstriert die Ausnutzung eines der im Abschnitt 4.1 auf Seite 68 gelisteten Vorteile der Modell-Middleware, einen Service durch einfache Abwesenheit des Plug-ins abschalten zu können.

A.4. Output-Plug-ins

Output-Plug-ins werden über den Typ `OutputPluginType` identifiziert – ein Synonym für den Typ `unsigned int`. Jedes Output-Plug-in registriert einen eigenen Alias.

Ein Output-Plug-in erzeugt eine Ausgabe für einen zuvor registrierten Parameter, wenn dessen post_write-Callback ausgelöst wurde. Die Erscheinung, d.h. Umfang und Formatierung der Ausgabe, ist abhängig vom entsprechenden Output-Plug-in-Typ.

Die Schnittstelle der Output-Plug-ins erlaubt unter anderem die folgenden Aktionen:

- GreenConfig-Parameter können auf unterschiedliche Weise der Beobachtung hinzugefügt (registriert) werden: einzelne Parameter (also Objekt), Parameternamen, Listen von Parameternamen oder unter Angabe einer GreenConfig-API, deren sämtliche Parameter hinzugefügt werden.
- Beobachtete Parameter können wieder entfernt werden.
- Output-Plug-ins können pausiert und wieder gestartet werden. Die Angabe eines SystemC-Events oder eines Simulations-Zeitpunktes erlaubt automatisches Starten.

Output-Plug-ins können pausiert starten, sodass sie erst bei end_of_elaboration automatisch mit der Ausgabe beginnen und (optional) zuvor aufgezeichnete Änderungen ausgeben[4].

Die folgenden Output-Plug-ins stehen zur Verfügung. Teilweise sind Visualisierungen ihrer Ausgaben durch externe Tools als Beispiel gezeigt.

Default-Output-Plug-in

Das Default-Output-Plug-in ist ein Spezialfall:
Es wird über den Alias `DEFAULT_OUT` (=0) identifiziert und wird durch ein zuvor vom Benutzer definiertes Output-Plug-in repräsentiert. Es ist der Default für alle Ausgaben in der gesamten Simulation, wenn nicht explizit ein anderes Output-Plug-in oder ein anderer Typ für die jeweilige Ausgabe angegeben ist.

NULL-Output-Plug-in

Auch das NULL-Output-Plug-in ist ein Spezialfall:
Es wird über den Alias `NULL_OUT` (=1) identifiziert und bedeutet kein Output-Plug-in. Durch

[4] Dieses Verhalten ist nützlich, wenn die Ausgabe erst begonnen bzw. initialisiert werden darf, wenn alle zu beobachtenden Parameter hinzugefügt wurden, z.B. bei einer sequenziell geschriebenen Tabelle, in deren Kopf die Parameternamen stehen sollen.

die Definition des Default-Output-Plug-ins auf dieses NULL-Output-Plug-in kann beispielsweise die Standardausgabe deaktiviert werden.

Standard-Output-Plug-in

Das Standard-Output-Plug-in gibt jede Änderung der beobachteten Parameter mit einem Zeitstempel als Zeile auf dem Standard-Output-Stream aus, typischerweise dem Terminal. Listing A.2 zeigt ein Beispiel. Das Standard-Output-Plug-in wird über den Alias `STDOUT_OUT` (=3) identifiziert.

```
@105 ns /4 (AnalysisPlugin.STDOUT_OUT_default): AVTool.param = 1
@110 ns /2 (AnalysisPlugin.STDOUT_OUT_default): MyModule.param = 0
@110 ns /2 (AnalysisPlugin.STDOUT_OUT_default): AVTool.param = 100
```

Listing A.2: Beispielausgabe des Standard-Output-Plug-ins

Text-Datei-Output-Plug-in

Das Text-Datei-Output-Plug-in schreibt jede Änderung der beobachteten Parameter als Zeile in eine Textdatei. Auch hier wird ein Zeitstempel mit ausgegeben. Listing A.3 zeigt ein Beispiel. Das Text-Datei-Output-Plug-in wird über den Alias `TXT_FILE_OUT` (=2) identifiziert.

```
Simulation time: Fri Jan  7 14:55:16 2011

@0 s /0: Owner.str_param = This is a test string.
@0 s /0: Owner.uint_param = 1
@1 ns /1: Owner.uint_param = 670
@1 ns /1: Owner.str_param = Hello World!
@2 ns /4: Owner.str_param = Hello Germany!
@3 ns /7: Owner.uint_param = 2000
@106 ns /10: Owner.int_param = 133
```

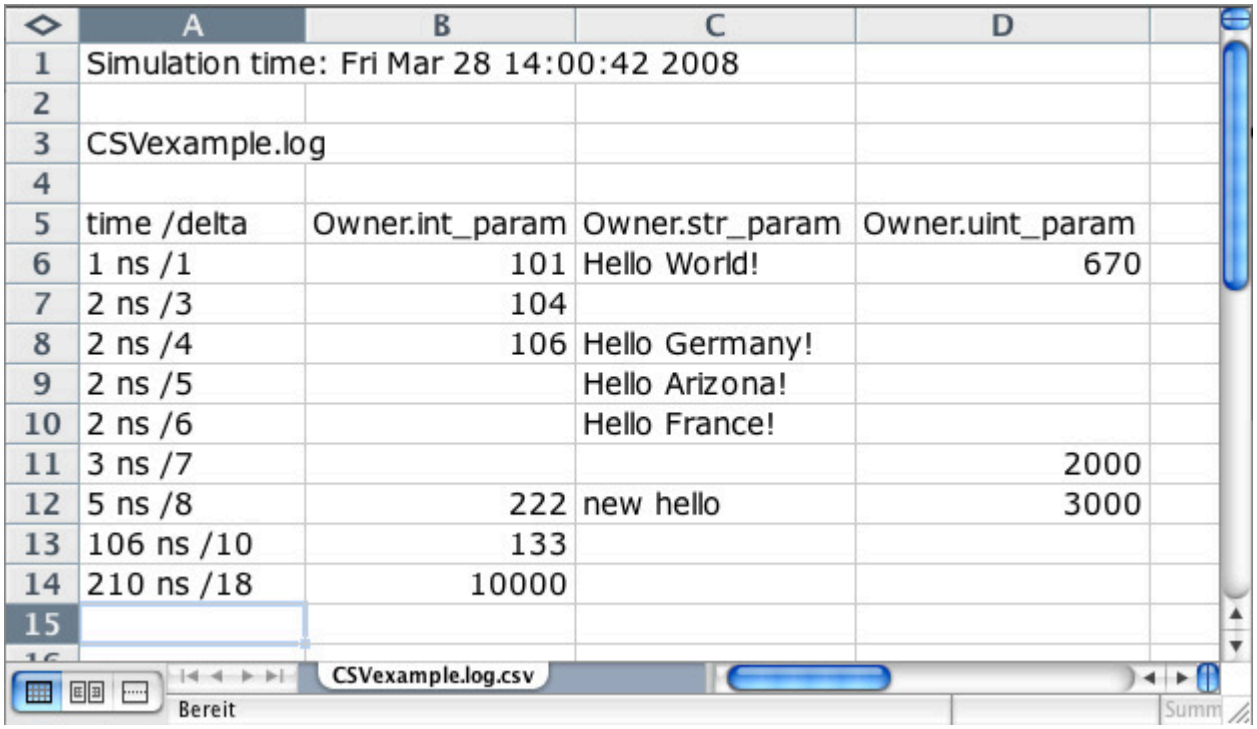
Listing A.3: Beispieldatei des Text-Datei-Output-Plug-ins

CSV-Datei-Output-Plug-in

Das CSV-Datei-Output-Plug-in (Comma-Sepatated Values (CSV)) erzeugt eine CSV-Datei [Shaf05][i], die beispielsweise mit Microsoft Excel geöffnet[5] und tabellarisch dargestellt werden kann (siehe Abbildung A.4). Damit die Parameternamen in die erste Zeile geschrie-

[5]Als Separator verwendet das Output-Plug-in Semikolons. Das kann konfiguriert werden.

ben werden können, sollte das Output-Plug-in erst gestartet werden[6], wenn alle Parameter zur Beobachtung hinzugefügt wurden. Das CSV-Datei-Output-Plug-in wird über den Alias `CSV_FILE_OUT` (=4) identifiziert.



	A	B	C	D
1	Simulation time: Fri Mar 28 14:00:42 2008			
2				
3	CSVexample.log			
4				
5	time /delta	Owner.int_param	Owner.str_param	Owner.uint_param
6	1 ns /1	101	Hello World!	670
7	2 ns /3	104		
8	2 ns /4	106	Hello Germany!	
9	2 ns /5		Hello Arizona!	
10	2 ns /6		Hello France!	
11	3 ns /7			2000
12	5 ns /8	222	new hello	3000
13	106 ns /10	133		
14	210 ns /18	10000		
15				

Abbildung A.4.: Anzeige einer vom CSV-Datei-Output-Plug-in erzeugten CSV-Datei mit MS Excel

VCD-Datei-Output-Plug-in

Das VCD-Datei-Output-Plug-in (Value Change Dump (VCD)) erzeugt eine VCD-Datei [IEEE01, S.325 ff.]. Dieses für die Hardwarebeschreibungssprache Verilog [IEEE01] entwickelte Format kann beispielsweise vom Tool GTKWave[7] wie in Abbildung A.5 dargestellt werden. Auch dieses Output-Plug-in sollte erst gestartet werden, wenn alle Parameter zur Beobachtung hinzugefügt wurden. Das VCD-Datei-Output-Plug-in wird über den Alias `VCD_FILE_OUT` (=6) identifiziert.

SCV-Stream-Output-Plug-in

Das SCV-Stream-Output-Plug-in erzeugt Transaktions-Streams nach dem SCV-Standard [OSCI03]^i. Diese Streams können von Tools dargestellt werden und zeigen jeweils den zeitlichen Verlauf eines Parameters. Tests mit dem CoWare Platform-Architect und Mentor Graphics ModelSim (vgl. Abbildung A.6) waren erfolgreich. Das Output-Plug-in kann kon-

[6] Das CSV-Datei-Output-Plug-in erzeugt die Datei bei der ersten Ausgabe.
[7] GTKWave ist ein Tool für die Anzeige von Signalverläufen: `http://gtkwave.sourceforge.net`

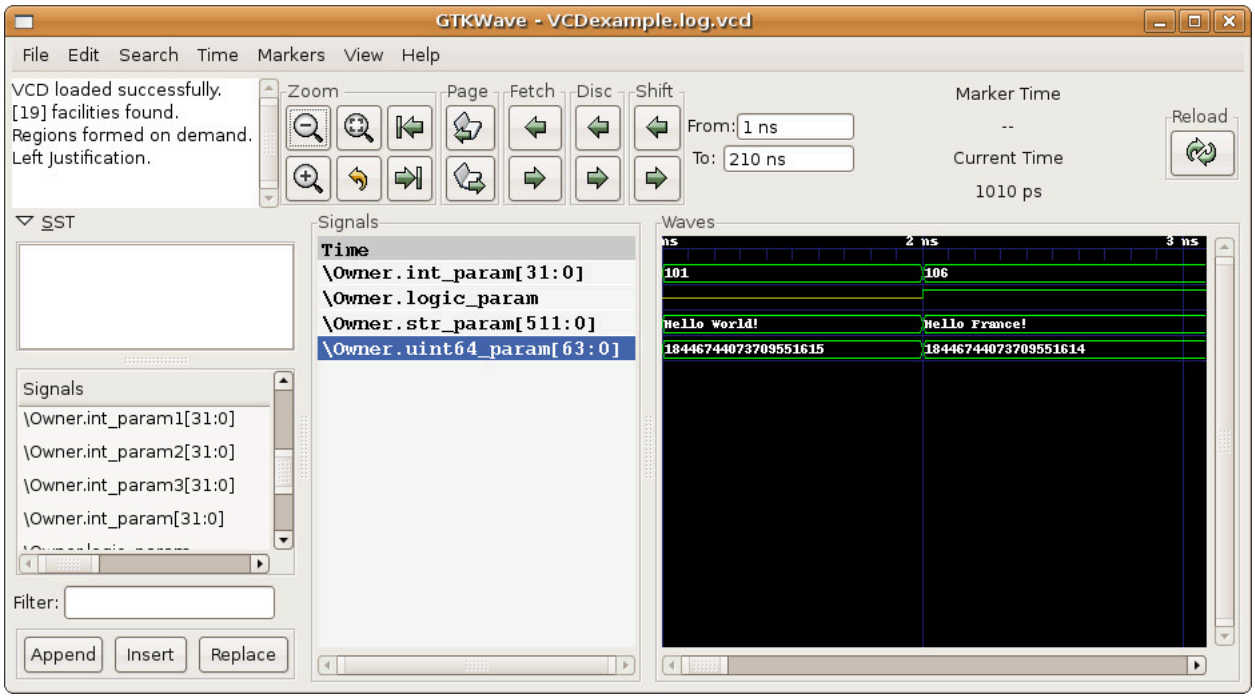


Abbildung A.5.: Anzeige einer vom VCD-Datei-Output-Plug-in erzeugten VCD-Datei mit GTKWave

figuriert werden, um jeweils kompatible Ausgaben zu erzeugen. Der Code zu Abbildung A.6 ist im Listing B.15 zu finden.

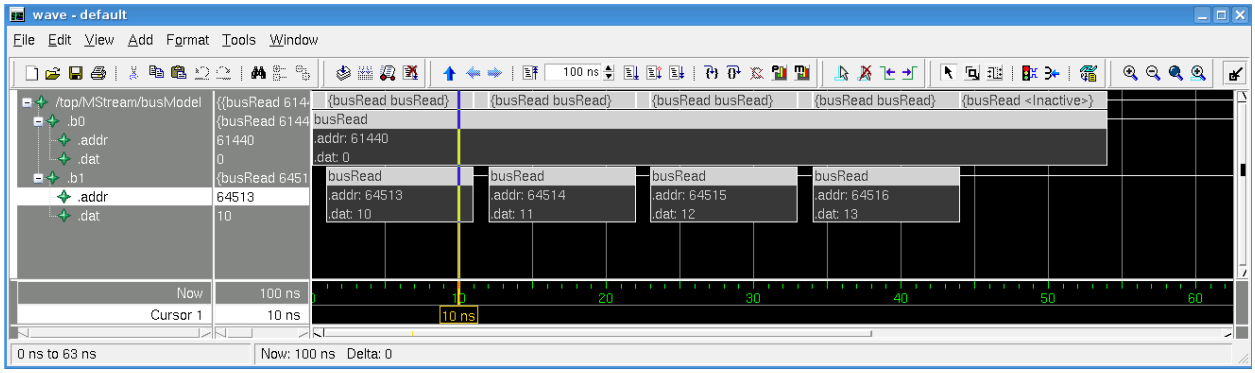


Abbildung A.6.: Anzeige eines vom SCV-Stream-Output-Plug-in erzeugten Streams in ModelSim

A.5. Analyse-Plug-in

Das Analyse-Plug-in ist das in Abschnitt A.1.1 vorgestellte zentrale Singleton-Objekt. Abbildung A.7 zeigt ein Klassendiagramm mit einer Übersicht. Es sind die für die Modell-Middleware benötigten Interfaces (`command_if`, `gc_port_if`) und die dafür implementierten Funktionen (`transport`, `getName`, `getCommandName`, `getCommandDescription`) zu erkennen. Das Default-Output-Plug-in kann entweder mit dem Konstruktor oder der Funktion

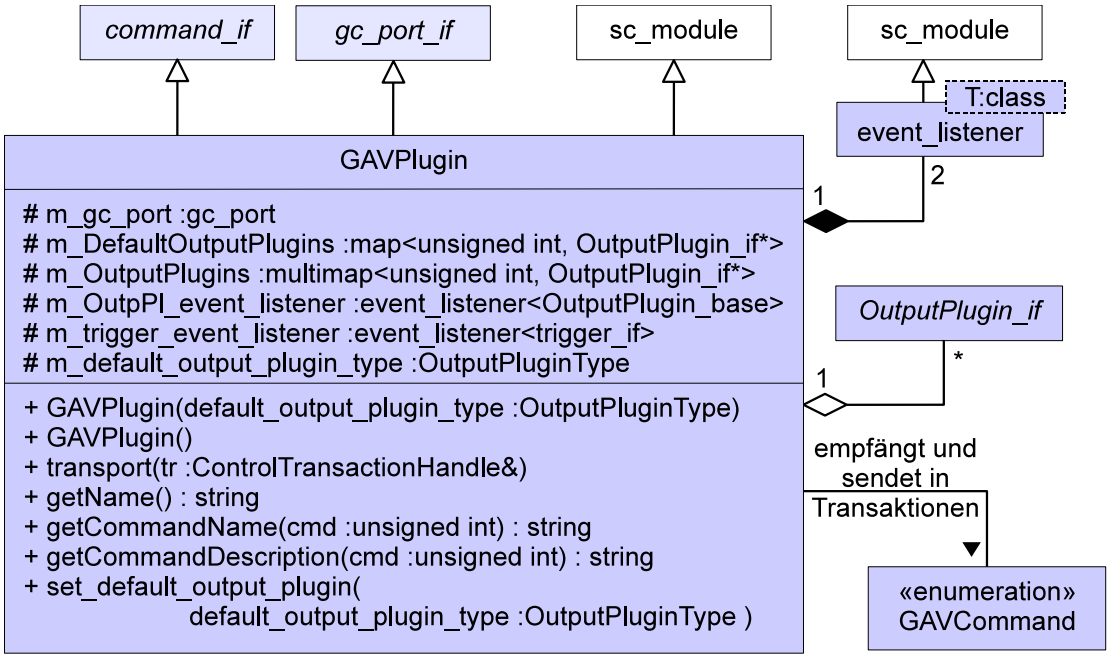


Abbildung A.7.: Übersicht Analyse-Plug-in (Klassendiagramm)

`set_default_output_plugin` festlegt werden und wird im Member `m_default_output_plugin` gespeichert.

Zwei assoziative Container (`m_DefaultOutputPlugins` und `m_OutputPlugins`) speichern die verschiedenen Output-Plug-ins. Die Event-Listener sind zwei weitere Members.

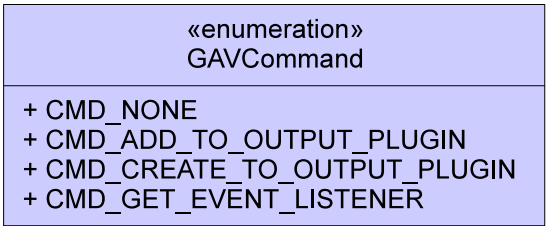


Abbildung A.8.: Analyse-Middleware-Kommandos

Abbildung A.8 listet die Middleware-Kommandos, die vom Analyse-Plug-in und den User-APIs verwendet werden. Sie ermöglichen die Zugriffe auf Output-Plug-ins und Event-Listener.

A.6. Statistik-Kalkulator

Im Folgenden werden Details zu der bereits in Abschnitt A.1.3 vorgestellten User-API Statistik-Kalkulator gegeben.

Zunächst existiert ein Statistik-Kalkulator, nachdem er vom Benutzer instanziiert wurde, weitgehend unabhängig vom Analyse-Service, hat also keine Verbindung zum Analyse-Plug-in. Lediglich für den Zugriff auf einen Event-Listener verfügt der enthaltene Trigger über eine GreenAV-API.

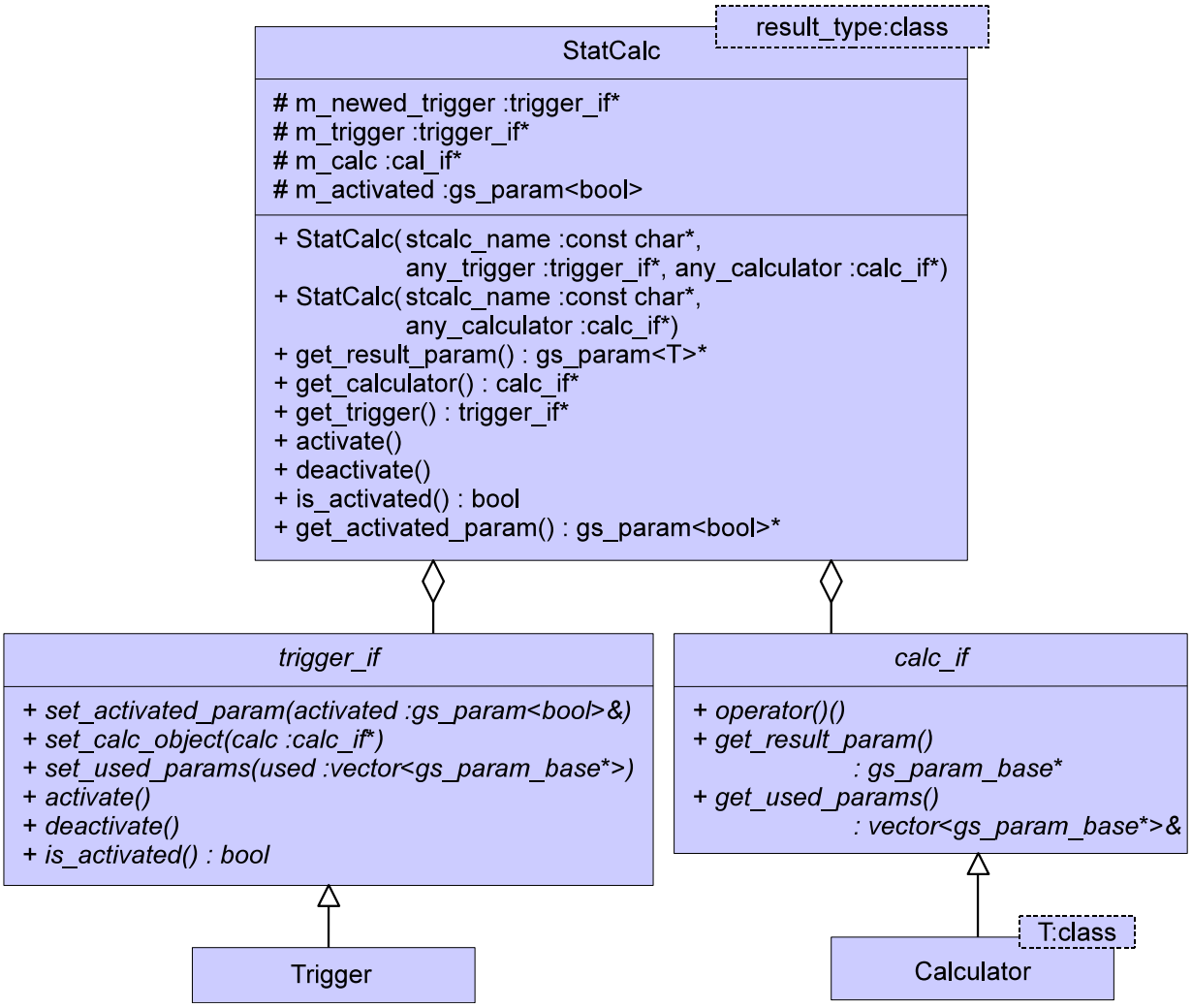


Abbildung A.9.: Statistik-Kalkulator (Klassendiagramm)

Abbildung A.9 zeigt das Klassendiagramm des Statistik-Kalkulators (Klasse `StatCalc`). Es ist ersichtlich, dass er (in den Members `m_trigger` und `m_calc`) jeweils einen Pointer auf eine Instanz eines Triggers und eines Kalkulators besitzt. Der Member `m_newed_trigger` zeigt auf das gleiche Objekt wie `m_trigger`, wenn der Trigger vom Statistik-Kalkulator selbst erzeugt wurde, also auch wieder gelöscht werden muss, sonst ist er Null.

Die Funktionen `get_trigger` und `get_calculator` ermöglichen Zugriff von außen auf die Members.

Ein Statistik-Kalkulator kann aktiviert und deaktiviert werden (vgl. Abschnitt A.6.2). Der aktuelle Status wird im Member-GreenConfig-Parameter `m_activated` gehalten und kann einerseits durch Setzen des Parameters, andererseits über die Funktionen `activate` und `deactivate` manipuliert und über die Funktionen `is_activated` und `get_activated_param` abgefragt werden.

Die Funktion `get_result_param` reicht den Ergebnisparameter vom Kalkulator durch. Dieser Parameter ist ein vom Kalkulator erstellter GreenConfig-Parameter, in den das Ergebnis geschrieben wird, sodass dieser Parameter beispielsweise von einem Output-Plug-in ausgegeben werden kann.

Ebenfalls in Abbildung A.9 dargestellt sind die beiden Interfaces `trigger_if` und `calc_if` und zwei jeweils davon abgeleitete Klassen `Trigger` und `Calculator`.

Die wesentlichen Aufgaben des Statistik-Kalkulators werden vom Trigger und Kalkulator übernommen, die jeder Statistik-Kalkulator besitzt:

A.6.1. Kalkulator

Ein Kalkulator muss vom Interface `calc_if` ableiten, damit der Statistik-Kalkulator die entsprechenden Aufrufe tätigen kann. Der Kalkulator kann vom Benutzer erzeugt und an den Konstruktor eines Statistik-Kalkulators übergeben werden. Das Interface definiert drei Funktionen (vgl. Abbildung A.9): Beim Aufruf des Klammeroperators, z.B. durch den Trigger, führt der Kalkulator die gewünschte Berechnung sofort durch. Die Interface-Funktion `get_result_param` liefert den GreenConfig-Parameter, der vom Kalkulator erstellt wurde und das Berechnungsergebnis enthält. Die Funktion `get_used_params` liefert einen Vektor über alle in der Berechnung verwendeten GreenConfig-Parameter. Dieser Vektor wird dem Trigger übergeben, um die Berechnung bei deren Änderungen auslösen zu können.

Der Analyse-Service stellt zwei Kalkulator-Klassen zur Verfügung: Der maßgebliche Kalkulator ist die Klasse `Calculator`. Ein weiterer (Klasse `Calculator_bit`) fügt diesem zusätzliche Bit-Operationen hinzu. Die Kalkulatoren werden auf einen Datentyp festgelegt, der den Typ der Zwischenergebnisse und des Ausgabeparameters festlegt und können dynamisch konfiguriert werden. Sie unterstützen mathematische Operationen ($+$, $-$, $*$, $/$), logische Operationen ($==$, $!=$, $>=$, $<=$, $>$, $<$) und bitweise Operationen (&, |).

Die Definition der Formel kann auf zwei verschiedene Weisen vorgenommen werden: eine vereinfachte und eine flexiblere („Calc-Syntax“), vgl. Listings A.10 und A.11 und für Details [ScKG10][i].

```
1 gs_param<double>  dbl_p("dbl_p") ;  gs_param<int>           int_p("int_p") ;
2 gs_param<int>    int_p2("int_p2");  gs_param<unsigned int> uint_p("uint_p");
3
4 Calculator<double> c1("my_calculator");
5 c1.calc("+", c1.calc("/", c1.calc("-", int_p, int_p2),
6                            int_p2),
7              c1.calc("*", dbl_p, uint_p));
```

Listing A.10: Kalkulator-Formel-Beispiel in Calc-Syntax
Formel: $(((int\_p - int\_p2)/int\_p2) + (dbl\_p * uint\_p))$.

In Formeln können an Stelle von Eingabeparameter auch Konstanten angegeben werden.

Eine weitere Fähigkeit der Kalkulatoren ist die optional zu aktivierende fehlertolerante Handhabung von Laufzeitfehlern in der Berechnung. Dann wird für jede Rechenoperation

```
gs_param_base &dbl_p_  = dbl_p ; // gs_param<double>
gs_param_base &int_p_  = int_p ; // gs_param<int>
gs_param_base &int_p2_ = int_p2; // gs_param<int>
gs_param_base &uint_p_ = uint_p; // gs_param<unsigned int>

gs::av::Calculator<int> cc("my_convCalc");
cc(cc(cc(int_p_ - int_p2_) / int_p2_) + cc(dbl_p_ * uint_p_));
```

Listing A.11: Kalkulator-Formel-Beispiel in vereinfachter Syntax
Formel: $(((int\_p - int\_p2)/int\_p2) + (dbl\_p * uint\_p))$.

im Fehlerfall ein Standardwert zurückgegeben[8]. Beispielsweise liefert die Division durch Null in diesem Modus Null, anstatt die Simulation mit einem Fehler zu beenden.

Eine bereits eingebaute Statistik-Fähigkeit ist ein Fenster, über das das arithmetische Mittel gebildet wird („Sliding-Window“).

Ein einfacher Erweiterungsmechanismus erlaubt das Hinzufügen von weiteren Rechenoperationen. Zu diesem Zweck definiert der Benutzer eine globale Template-Funktion mit dem Template-Parameter `T` und der Signatur `name(inA :T, inB :T, sloppy :bool) : T` und registriert diese beim Kalkulator mit der Funktion `addFunc`.

Weitere Merkmale, wie beispielsweise umfangreiche Statistiken, können vom Benutzer erweitert werden, indem er entweder den vorhandenen Kalkulator modifiziert oder neue Kalkulatoren entwickelt. Einzige Voraussetzung ist das Implementieren des `calc_if`-Interfaces.

A.6.2. Trigger

Ein Statistik-Kalkulator benötigt einen Trigger, um die Berechnung im Kalkulator zu den gewünschten Zeitpunkten und Ereignissen auszulösen. Ein Trigger kann optional vom Benutzer konfiguriert und dem Statistik-Kalkulator übergeben werden. Erfolgt dies nicht, erzeugt der Statistik-Kalkulator selbstständig einen Trigger mit Standardverhalten.

Um in einem Statistik-Kalkulator verwendet werden zu können, muss ein Trigger vom Interface `trigger_if` ableiten (vgl. Abbildung A.9 auf Seite 183) und damit die folgenden Funktionen implementieren: Die Funktionen `activate` und `deactivate` dienen dem Aktivieren und Deaktivieren des Triggers. Ist ein Trigger deaktiviert, hat er alle GreenConfig-Parameter-Callbacks deaktiviert, sodass er keinen Performance-Overhead mehr verursacht. Die Funktion `is_activated` liefert den aktuellen Aktivierungsstatus. Mit der Funktion `set_activated_param` kann der Statistik-Kalkulator dem Trigger einen GreenConfig-Parameter übergeben, den dieser einerseits stets mit seinem Aktivierungsstatus aktuell halten und andererseits Änderungen an diesem in seinen Aktivierungsstatus übernehmen muss. Die

[8] Außerdem wird eine Warnung ausgegeben.

Funktion `set_used_params` wird verwendet, um dem Trigger sämtliche relevanten Eingabeparameter zu übergeben[9].

Der vom Analyse-Service bereitgestellte Trigger (Klasse `Trigger`) registriert für alle Eingabeparameter post_write- und destroy_param-Callbacks (vgl. Seite 95).

Der Trigger unterstützt die folgenden Auslöseereignisse, die im Konstruktor oder durch Funktionsaufrufe angegeben werden:

- Das *Standardverhalten* ist das Auslösen der Berechnung bei jedem *post_write-Callback* aller Eingabeparameter.
- Alternativ zum Standardverhalten kann dem Trigger ein beliebiger GreenConfig-Parameter als *Aktivator* übergeben werden, bei dessen Änderung (Callback) der Trigger auslöst.
- Dem Trigger kann ein *SystemC-Event* übergeben werden. Wenn es auslöst, löst auch der Trigger die Berechnung aus.
- Eine *periodische Auslösung* kann durch Angabe einer Zeitspanne (`sc_time`) realisiert werden.
- *Manuelle Auslösung* kann durch Aufruf der Funktion `calculate_now` erreicht werden.
- Dem Trigger kann ein *Wächter-Parameter* vom Typ `gs_param<bool>` für eine boolsche Vorbedingung übergeben werden. Die Auslösung nach dem eingestellten Ereignis wird nur durchgeführt, wenn dieser Wächter wahr liefert.

Benutzer können alternative Trigger mit abweichendem Verhalten implementieren. Dabei ist lediglich auf die Ableitung vom Interface `trigger_if` zu achten.

[9] Die Eingabeparameter stammen aus dem Kalkulator.

B. Externe Listings

Die hier aufgeführten Listings können den jeweils angegebenen externen Quellen entnommen werden. Wie in der Einleitung im Abschnitt 1.3 bereits beschrieben, handelt es sich meist um größere Code-Beispiele.

Listing B.1: Das Beispiel für Serviceerweiterungen befindet sich in [Schr11][i] im Verzeichnis greencontrol/examples/new_service_example

Listing B.2: Das hierarchy_precedence-Beispiel für hierarchische Rangfolge in einem Mechanismus zeitlicher Rangfolge befindet sich in [Schr11][i] im Verzeichnis greencontrol/examples/regression_tests/hierarchy_precedence_tests

Listing B.3: Das Beispiel für die Nutzung des Analyse-Services befindet sich in [Schr11][i] im Verzeichnis greencontrol/examples/gav_demonstration_platform

Listing B.4: Das scml_example-Beispiel befindet sich in [Schr11][i] im Verzeichnis greencontrol/examples/scml_example und in [Schr11b] im Code-Verzeichnis Workspaces/CoWareWorkspace/scml_example

Listing B.5: Das wiz_scml_example-Beispiel befindet sich in [Schr11b] im Code-Verzeichnis Workspaces/CoWareWorkspace/wiz_models/wiz_scml_example

Listing B.6: Das wiz_scml_example-Beispiel befindet sich in [Schr11b] im Code-Verzeichnis Workspaces/CoWareWorkspace/gc_example_wrapped_to_coware

Listing B.7: Das ccss_gs_param-Beispiel befindet sich in [Schr11b] im Code-Verzeichnis Workspaces/SynopsysWorkspace/ccss_gs_param. Verschiedene Beispiel-Simulationen im Workspace machen davon Verwendung.

Listing B.8: Das only_ccss_param_als_gs_param-Beispiel befindet sich in [Schr11b] im Code-Verzeichnis Workspaces/SynopsysWorkspace/only_ccss_param_als_gs_param

Listing B.9: Das CASI-IP_in_GreenConfig_A1_OSCI-Beispiel ist aus lizenzrechtlichen Gründen nicht öffentlich verfügbar.

Listing B.10: Das CASI-IP_in_GreenConfig_A1_OSCI-Beispiel ist aus lizenzrechtlichen Gründen nicht öffentlich verfügbar.

Listing B.11: Das CASI-Sim_mit_GCnf_IP-Beispiel ist aus lizenzrechtlichen Gründen nicht öffentlich verfügbar.

Listing B.12: Der modifizierte OVM-Code ist aus lizenzrechtlichen Gründen nicht öffentlich verfügbar.

Listing B.13: Das große Beispielsystem befindet sich in [Schr11b] im Code-Verzeichnis Workspaces/CoWareWorkspace/grosse_MixedPlatform. Die Unterverzeichnisse build_OSCI und build_CoWare enthalten jeweils die Projektdateien für die beiden Demonstrationen.

Listing B.14: Der Code für SCRSI befindet sich in [Schr11b] im Code-Verzeichnis SCRSI. Anwendungsbeispiele sind dort im Verzeichnis example/gav_demonstration_platform und im externen Listing B.13 zu finden.

Listing B.15: Das gav_transaction_scv_example-Beispiel befindet sich in [Schr11b] im Code-Verzeichnis Workspaces/ModelSim/wiz_models/gav_transaction_scv_example

Listing B.16: Das realistic_with_GSGPSocket-Beispiel befindet sich in [Schr11b] im Code-Verzeichnis greencontrol_workspace/performance_test

Verzeichnisse

Abkürzungen

API	Application Programming Interface, Programmierschnittstelle
CASI	Cycle Accurate Simulation Interface
CCI	Configuration, Control & Inspection
CCI WG	Configuration, Control & Inspection Working Group
CSV	Comma-Sepatated Values
DRF	Device Register Framework
ESL	Electronic System Level
GUI	grafische Benutzeroberfläche
IDE	integrierte Entwicklungsumgebung
IP	Intellectual Property
OSCI	Open SystemC Initiative
OVM	Open-Verification-Methodology
PIU	Proprietäres Modell in universeller Entwicklungsumgebung
RTL	Register-Transfer-Ebene
SimPh	Simulationsphase
SCML	SystemC-Modeling-Library
SCRSI	SystemC-Remote-Service-Interface
SCV	SystemC Verification Standard
TLM	Transaction-Level-Modellierung
UIP	Universelles Modell in proprietärer Entwicklungsumgebung
VCD	Value Change Dump

Stichwortverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Literatur

[AlEH06] Albrecht, Carsten ; Eibl, Christian J. ; Hagenau, Rainer: A Loosely-Coupled Graphical User Interface for Run-Time Control of SystemC Simulation Models. In: *International Journal of Simulation Systems, Science & Technology* Bd. 7 Nr. 3, 2006, S. 1–11

[ARA+07] Albertini, Bruno ; Rigo, Sandro ; Araujo, Guido ; Araujo, Cristiano ; Barros, Edna ; Azevedo, Willians: A computational reflection mechanism to support platform debugging in SystemC. In: *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY : ACM, 2007, S. 81–86

[ARM06] ARM Ltd.: *Cycle-Accurate Simulation Interface (CASI)*. V1.1.0. Cambridge, 2006

[ARM07] ARM Ltd.: *ARM RealView ESL API*. V2.0. Cambridge, 2007

[BaMA07] Bailey, Brian ; Martin, Grant ; Andrew, Piziali: *ESL Design and Verification - A Prescription for Electronic System-Level Methodology*. 1. Aufl. San Francisco, CA : Morgan Kaufmann, 2007

[BBF+08] Beltrame, Giovanni ; Bolchini, Cristiana ; Fossati, Luca ; Miele, Antonio ; Sciuto, Donatella: ReSP: a non-intrusive transaction-level reflective MPSoC simulation platform for design space exploration. In: *ASP-DAC '08: Proceedings of the 2008 Asia and South Pacific Design Automation Conference*. Los Alamitos, CA : IEEE Computer Society Press, 2008, S. 673–678

[BeAl07] Berrada, Marouane ; Aldis, James: SystemPython: a Python extension to control SystemC SoC simulations. In: *GreenSocs Meeting Presentation, Design and Test in Europe Conference (DATE)*. Nizza, 2007

[Cade09] Cadence Design Systems, Inc.: *OVM-SC Library Reference*. V2.0.1. San José, CA, 2009

[Cade10] Cadence Design Systems, Inc.: *OVM Multi-Language Methodology*. V2.1.1. San José, CA, 2010

[CaGa03] Cai, Lukai ; Gajski, Daniel: Transaction level modeling: an overview. In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY : ACM, 2003, S. 19–24

[ChMa05] Charest, L. ; Marquet, P.: Comparisons of different approaches of realizing IP block configuration in SystemC. In: *IEEE-NEWCAS Conference, 2005. The 3rd International*. New York, NY, 2005, S. 83 – 86

[CoWa09] CoWare Inc.: *SystemC Modeling Library Manual*. V1.4. San José, CA, 2009

[Gers01] Gerstlauer, Andreas: *The SpecC Methodology*. Center for Embedded Computer Systems, University of California, Irvine, 2001

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA : Addison-Wesley Professional, 1994

[GLGS02] Grötker, Thorsten ; Liano, Stan ; Grant, Martin ; Swan, Stuart: *System Design with SystemC*. Norwell, MA : Kluwer Academic Publishers, 2002

[Golz09] Golze, Ulrich: *Skript zur Vorlesung Chip- und System-Entwurf II*. Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), 2009

[Günz11] Günzel, Robert: *Taktgenaue Bus-Simulation mit der Transaction-Level-Modellierung*, Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Dissertation, 2011

[IEEE01] Institute of Electrical & Electronics Engineers, Inc.: IEEE Standard Verilog Hardware Description Language. In: *IEEE Std 1364-2001*. New York, NY, 2001

[IEEE06] Institute of Electrical & Electronics Engineers, Inc.: IEEE Standard SystemC Language Reference Manual. In: *IEEE Std 1666-2005*. New York, NY, 2006

[IEEE08] Institute of Electrical & Electronics Engineers, Inc.: IEEE Standard for the Functional Verification Language E. In: *IEEE STD 1647-2008*. New York, NY, 2008

[IEEE09] Institute of Electrical & Electronics Engineers, Inc.: IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language. In: *IEEE STD 1800-2009*. New York, NY, 2009

[Ieru06] Ierusalimschy, Roberto: *Programmieren mit Lua.* München : Open Source Press, 2006

[ISO03] International Organization for Standardization: Programming Language C++. In: *ISO/IEC 14882:2003.* Genf, 2003

[Jesk10] Jeske, Dimitri: *Erweiterung des SystemC-Remote-Service-Interface-Frameworks mit Parameter-und Breakpoint-Services*, Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Studienarbeit, 2010

[Klin08] Klingauf, Wolfgang: *Systematic Transaction-Level Communication Modeling with SystemC*, Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Dissertation, 2008

[Meie08] Meier, Gerrit: *Service-orientierte Kontrolle und Visualisierung von SystemC-Simulationen unter Verwendung der Kontrol- und Analyseframeworks GreenControl und GreenAV*, Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Diplomarbeit, 2008

[MES+10] Montón, Màrius ; Engblom, Jakob ; Schröder, Christian ; Carrabina, Jordi ; Burton, Mark: Checkpoint and Restore for SystemC Models. In: Borrione, Dominique (Hrsg.): *Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's. Selected Contributions on Specification, Design, and Verification from FDL 2009.* Niederlande : Springer, 2010, S. 41–57

[Mont10] Montón, Màrius: *Checkpointing for Virtual Platforms and SystemC-TLM-2.0*, Universitat Autònoma de Barcelona. Escola d'Enginyeria Departament de Microelectrònica i Sistemes Electrònics, Dissertation, 2010

[OSCI09] Open SystemC Initiative (OSCI): *OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL.* JA32. Sunnyvale, CA, 2009

[PaHe08] Patterson, David A. ; Hennessy, John L.: *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design).* San Francisco, CA : Morgan Kaufmann Publishers Inc., 2008

[PMBS06] Patel, Hiren D. ; Mathaikutty, Deepak A. ; Berber, David ; Shukla, Sandeep K.: CARH: Service-Oriented Architecture for Validating System-Level Designs. In: *IEEE Transactions on CAD of Integrated Circuits and Systems* Bd. 25 Nr. 8. New York, NY, 2006, S. 1458–1474

[RGDR07] Rogin, Frank ; Genz, Christian ; Drechsler, Rolf ; Rülke, Steffen: An Integrated SystemC Debugging Environment. In: *FDL '07: Forum on Specification & Design Languages*. Barcelona, 2007, S. 140–145

[Ruet10] Rütz, Michael: *Ein generischer Transaction-Logger für die Modell-Middleware GreenControl*, Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Diplomarbeit, 2010

[Schm11] Schmidt, Alexander: *Entwurf eines Prä-Simulations-Servicekonzepts für das SystemC-Entwurfswerkzeug SCRSI*, Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Studienarbeit, 2011

[Schr11a] Schröder, Christian: Configuration Interoperability of Hardware-Software-Models in SystemC. In: *DATE '11: Proceedings of the Design, Automation & Test in Europe, PhD Forum*. Heverlee, Leuven : EDAA, 2011

[Schr11b] Schröder, Christian: *Praktische Untersuchungen zu Meta-Interoperabilität*, Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Forschungsbericht, 2011

[SKG+09] Schröder, Christian ; Klingauf, Wolfgang ; Günzel, Robert ; Burton, Mark ; Roesler, Eric: Configuration and control of SystemC models using TLM middleware. In: *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY : ACM, 2009, S. 81–88

[SPIR08] Spirit Consortium: *IP-XACT v1.4: A specification for XML meta-data and tool interfaces*. Napa, CA, 2008

[Tutt09] Tuttas, Jan: *Generische Inter-Service-Kommunikation für das SystemC-Remote-Service-Interface SCRSI*, Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Diplomarbeit, 2009

Internetquellen

[ARM08] ARM Ltd.: *Carbon Design Systems Acquires SoC Designer From ARM.* Version: 2008. http://www.arm.com/about/newsroom/21628.php, Abruf: 04. Juni 2010

[Bart10] Bartholomeu, Marcus ; GreenSocs Ltd. (Hrsg.): *GreenScript Projekt.* Version: 2010. http://www.greensocs.com/Projects/GreenScript, Abruf: 10. März 2011

[Cade10] Cadence Design Systems, Inc.: *OVM Multi-language Release 2.1.1.* Version: 2010. http://www.ovmworld.org/contributions-details.php?id=40&keywords=OVM_Multi-language_Release_2.1.1, Abruf: 26. Mai 2010

[CoWa10a] CoWare, Inc.: *CoWare Platform Architect.* Version: 2010. http://www.coware.com/PDF/products/1.10.PlatformArchitect.pdf, Abruf: 04. März 2010

[CoWa10b] CoWare, Inc.: *SCML Source Code Kit.* Version: 2010. http://www.coware.com/solutions/scml_kit.php, Abruf: 02. Februar 2010

[ITRS10a] ITRS: *International Technology Roadmap for Semiconductors 2010 Update Overview.* Version: 2010. http://www.itrs.net/Links/2010ITRS/Home2010.htm, Abruf: 07. Mai 2011

[ITRS10b] ITRS: *More-than-Moore.* Version: 2010. http://www.itrs.net/Links/2010ITRS/IRC-ITRS-MtM-v23.pdf, Abruf: 07. Mai 2011

[OCPI10] Open Core Protocol International Partnership: *OCP-IP Provides Virtual Platform Leveraging Advanced OCP SystemC TLM Modeling Kit.* Version: 23. August 2010. http://www.ocpip.org/uploads/dynamic_areas/N2FqKmAYdInzJUciX09u/6189/VirtualPlatform_PressRelease_FNL.pdf, Abruf: 09. März 2011

[OSCI03] Open SystemC Initiative: *SystemC Verification Standard Specification.* Version: 1.0e, 2003. http://www.systemc.org/members/download_files/check_file?agreement=systemc_verification_library, Abruf: 17. Dezember 2010

[OSCI09] Open SystemC Initiative: *Requirements Specification for Configuration Interfaces*. Version: 1.0, 2009. `http://www.systemc.org/members/download_files/check_file?agreement=cci_config_rqmnts_091218`, Abruf: 19. Juli 2010

[OSCI10] Open SystemC Initiative: *Open SystemC Initiative Announces Public Review for Configuration Requirements of Configuration, Control & Inspection (CCI) Standardization Effort*. Version: 22. Februar 2010. `http://www.systemc.org/news/pr/view?item_key=dc5d930e51b8a23a32f120acf07bb804d2b286ec`, Abruf: 20. Juli 2010

[Patr07] Patrick, Jason B.: Device & Register Framework: Technology & API Overview. In: *Online presentation material, available at http://www.scribd.com/doc/6316145/Drf-Introduction* (2007)

[Pont10] Pontifícia Universidade Católica do Rio de Janeiro: *The Programming Language Lua*. Version: 2010. `http://www.lua.org`, Abruf: 15. Dezember 2010

[Schr10a] Schröder, Christian ; GreenSocs Ltd. (Hrsg.): *GreenReg Projekt-Quellen*. Version: 2010. `https://svn.greensocs.com/public/packages/greenreg/`, Abruf: 10. Mai 2011

[Schr10b] Schröder, Christian ; GreenSocs Ltd. (Hrsg.): *GreenReg User's Guide (GreenReg 4.0.x)*. Version: 2010. `http://www.greensocs.com/en/Projects/GreenReg/docs/GreenRegUsersGuide`, Abruf: 10. Mai 2011

[Schr11] Schröder, Christian ; GreenSocs Ltd. (Hrsg.): *GreenControl Projekt-Quellen*. Version: 2011. `https://svn.greensocs.com/public/packages/greencontrol/`, Abruf: 10. Mai 2011

[ScKG08] Schröder, Christian ; Klingauf, Wolfgang ; Günzel, Robert ; GreenSocs Ltd. (Hrsg.): *GreenAV Tutorial of use (GreenAV v.1.1.0)*. Version: 2008. `http://www.greensocs.com/en/Projects/GreenControl/GreenAV/docs/GreenAVTutorial`, Abruf: 10. Mai 2011

[ScKG10] Schröder, Christian ; Klingauf, Wolfgang ; Günzel, Robert ; GreenSocs Ltd. (Hrsg.): *Green Analysis and Visibility User's Guide (GreenAV v.4.2.0)*. Version: 2010. `http://www.greensocs.com/en/Projects/GreenControl/GreenAV/docs/GAVUsersGuide`, Abruf: 10. Mai 2011

[ScKl11a] Schröder, Christian ; Klingauf, Wolfgang ; GreenSocs Ltd. (Hrsg.): *GreenConfig User's Guide (GreenConfig 4.3.0)*. Version: 2011. `http://www.greensocs.com/en/Projects/GreenControl/GreenConfig/docs/GCnfUsersGuide`, Abruf: 10. Mai 2011

[ScKl11b] Schröder, Christian ; Klingauf, Wolfgang ; GreenSocs Ltd. (Hrsg.): *GreenControl User's Guide (GreenControl v.4.3.0)*. Version: 2011. http://www.greensocs.com/en/Projects/GreenControl/docs/GCUsersGuide, Abruf: 10. Mai 2011

[Shaf05] Shafranovich, Yakov: *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180, IETF. http://tools.ietf.org/html/rfc4180. Version: 2005, Abruf: 07.01.2011

[Syno08] Synopsys, Inc.: *Datasheet Innovator*. Version: 2008. http://www.synopsys.com/tools/sld/virtualplatforms/pages/innovator.aspx, Abruf: 13. Juli 2009

[Syno11] Synopsys, Inc.: *VPOM-3430*. Version: 2011. http://www.synopsys.com/Systems/VirtualPrototyping/VPModels/PreAssembled/Pages/VPOM-3430.aspx, Abruf: 07. Mai 2011

[Texa11] Texas Instruments: *OMAP 3430 - OMAP Wireless Processor Platform*. Version: 2011. http://focus.ti.com/general/docs/wtbu/wtbugencontent.tsp?templateId=6123&navigationId=12013&contentId=28741, Abruf: 07. Mai 2011

Lebenslauf

Name: Christian Schröder
Geboren: am 20. Januar 1981 in Birkenfeld (Nahe)
Familienstand: ledig

Schulbildung

1987 – 1991	Paul-Gerhardt-Schule (Grundschule), Meppen
1991 – 1993	Kardinal-von-Galen-Schule (Orientierungsstufe), Meppen
1993 – 2000	Windthorst-Gymnasium Meppen
	Abitur

Zivildienst

2000 – 2001	Zivildienst beim Paritätischen Emsland in Meppen

Studium

2001 – 2007	Informations-Systemtechnik (Diplom) an der TU Braunschweig
	Dipl.-Ing. Informations-Systemtechnik
2011	**Promotion Dr.-Ing.**

Berufserfahrung und Praktikum

2006	Praktikum bei der
	IAV GmbH Ingenieursgesellschaft Auto und Verkehr in Gifhorn,
	Abteilung Powertrain Mechatronik Entwicklung Dieselmotoren
2007 – 2011	Wissenschaftlicher Mitarbeiter
	Abteilung Entwurf integrierter Schaltungen (E.I.S.), Prof. Ulrich Golze
	TU Braunschweig